VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GLOBÁLNE OSVETLENIE V REÁLNOM ČASE

GLOBAL ILLUMINATION IN REAL-TIME

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MATEJ KARAS**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Karas Matej, Bc.**
Program: Informační technologie
Obor: Počítačová grafika a interakce
Název: **Globální osvětlení v reálném čase**
Global Illumination in Real-Time
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte techniky výpočtu globálního osvětlení v reálném čase a API Vulkan.
2. Vyberte vhodnou metodu výpočtu globálního osvětlení a navrhněte její rozšíření.
3. Implementujte navrženou metodu s využitím akcelerace na GPU.
4. Proměřte navrženou metodu a zhodnoťte ji.
5. Vytvořte demostrační video.

Literatura:

- Zander Majercik et al., "Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields", Journal of Computer Graphics Techniques Vol. 8, No. 2, 2019, ISSN 2331-7418

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2 a kostra aplikace.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Milet Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 30. října 2020

Abstrakt

Práca sa zaoberá fotorealistickým zobrazovaním a výpočtom globálneho osvetlenia v reálnom čase. V práci sú preskúmané metódy používané na výpočet globálnej iluminácie v reálnom čase, z ktorých bola vybratá state of the art metóda – Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields, ktorá na výpočet osvetlenia používa hardvérovú podporu sledovania lúčov. Použitie hardvérového sledovania lúčov vyžaduje novú generáciu grafických API a pre prácu bol vybraný Vulkan.

Abstract

This thesis deals with photorealistic rendering and real-time global illumination. Thesis contains overview of algorithms used for real-time global illumination of which the Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields was implemented. This algorithm uses hardware accelerated raytracing to compute global illumination in a scene. Hardware raytracing requires use of new generation of graphics API from which Vulkan was choosen for this thesis.

Kľúčové slová

RTX, globálne osvetlenie, Vulkan, Irradiance fields, PBR

Keywords

RTX, global illumination, Vulkan, Irradiance fields, PBR

Citácia

KARAS, Matej. *Globálne osvetlenie v reálnom čase*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet

Globálne osvetlenie v reálnom čase

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Tomáša Mileta. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Matej Karas
17. mája 2021

Poďakovanie

Ďakujem pánovi Ing. Tomášovi Miletovi za vedenie tejto práce a poskytnuté konzultácie. Taktiež vďaka rodine a všetkým, ktorý mi poskytli podporu v čase písania tejto práce.

Obsah

Kapitola 1

Úvod

Táto práca sa zaoberá výpočtom fotorealistického globálneho osvetlenia v reálnom čase, ktoré je v dnešnej dobe žiadané v hernom a filmovom priemysle. Dosiahnutie fotorealistického výsledku v reálnom čase, resp. interaktívnych snímkoch za sekundu je veľmi náročné až nemožné, a preto je nutné pristupovať k rôznym aproximáciám nutným pre interaktívnosť, čo však vedie k fyzikálnej inkorektnosti na úkor výkonu.

Pre dosiahnutie fotorealistického dojmu je výpočet nepriameho (ambientného) osvetlenia nutnosťou. Toto ambientné, nepriame osvetlenie vzniká neustálym rozptylovaním priameho osvetlenia v scéne. V dobe písania tejto práce je výkon počítačov natoľko dostatočný, že simulácia týchto javov nepredstavuje až tak veľký problém ako v minulosti.

V kapitole 2 sú popísané fyzikálne veličiny popisujúce chovanie svetla a následne fyzikálne založené vykresľovanie, ktoré je momentálny *state of the art*.

V kapitole 3 sú vymenované vybrané metódy, ktoré sú používané na výpočet globálnej iluminácie v reálnom čase. Pre túto prácu bola vybraná metóda *Dynamic diffuse global illumination with raytraced irradiance fields*, ktorá je momentálny *state of the art* v globálnej iluminácii, a ktorá na výpočet globálneho osvetlenia využíva novú technológiu – hardvérový *ray tracing*. Táto metóda dokáže naviac počítať viacero odrazov dynamickej globálnej iluminácie v reálnom čase – niečo, čo v minulosti nebolo možné.

V ďalšej kapitole 4 sú popísané vybrané časti návrhu aplikácie, následne v kapitole 5 sú popísané implementačné detaily a práca s Vulkánom, resp. *ray tracingom*, ktorý sa javí ako budúcnosť počítačovej grafiky. Na záver v kapitole 6 sú popísané uskutočnené merania, testy a zhodnotené dosiahnuté výsledky.

Kapitola 2

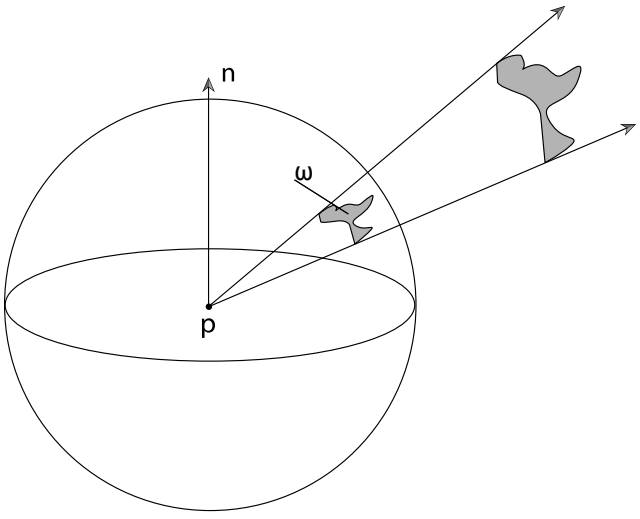
Fotorealistické zobrazovanie

Táto kapitola popisuje úvod do fotometrie, resp. popis rádiometrických veličín, ktoré sú používané pri výpočtoch svetelného žiarenia, potrebných pre pochopenie nasledujúcej sekcie, ktorá sa zaoberá *state of the art* technikou na výpočet fyzikálne založeného osvetlenia.

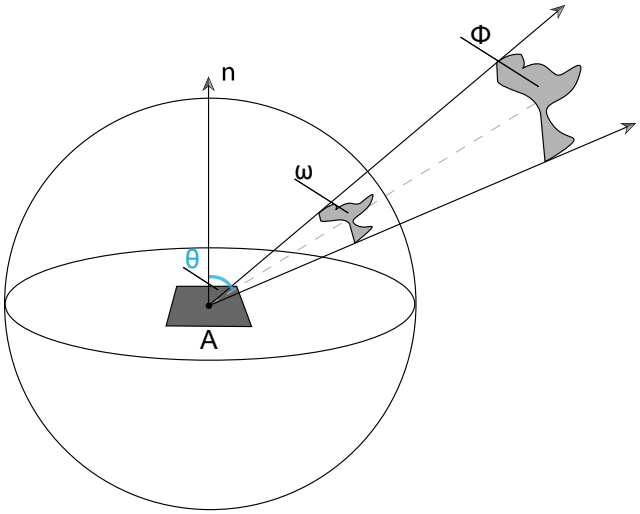
2.1 Fotometria

Fotometria je veda, ktorá sa zaoberá skúmaním pôsobenia svetelného žiarenia na ľudské oko. Pre jednoduchosť môžeme svetlo považovať za čisto elektromagnetické žiarenie s určitou frekvenciou, resp. určitou vlnovou dĺžkou a amplitúdou. Preto môžeme svetlo charakterizovať výkonom na jednotkovú plochu.

Biele svetlo (napr. slnečné žiarenie) sa skladá zo spektra vlnových dĺžok viditeľného svetla v intervale od $390nm$ po $700nm$. Po dopade na objekt je časť spektra pohltená a premenená na tepelnú energiu, (prípadne iný druh energie) a časť odrazená na základe jeho materiálových vlastností. Pozorovaná farba je následne spektrum odrazených vlnových dĺžok.



Obr. 2.1: Priestorový uhoľ ω udáva plochu nejakého objektu premietnutého na jednotkovú guľu. Inšpirované z [35].



Obr. 2.2: Žiara udáva energiu, ktorá je prítomná na ploche A skrz priestorový uhol ω a za prítomnosti žiarivého toku Φ.

Pri fyzikálnom výpočte svetla sa používajú rádiometrické veličiny. Rádiometrické jednotky majú ekvivalentné fotometrické jednotky (napr. žiara a svietivosť, svietivý tok a žiarivý tok, ...). Nižšie sú uvedené niektoré rádiometrické veličiny, s ktorými sa ďalej pracuje v tejto práci:

Priestorový uhol (solid angle) $\omega[sr]$ — priestorový uhol rozširuje jednotkovú kružnicu na jednotkovú guľu a udáva plochu objektu, premietnutého na túto guľu (obr. 2.1). Zjednodušene povedané, priestorový uhoľ popisuje *smer s objemom*.

Žiara (radiance) $L_e[W \cdot sr^{-1} \cdot m^{-2}]$ — energia prítomná na ploche A, určená priestorovým uhlom tvoreným premietnutou plochou svetelného zdroja (obr. 2.2). Hodnotu žiare je možné vypočítať podľa rovnice 2.1:

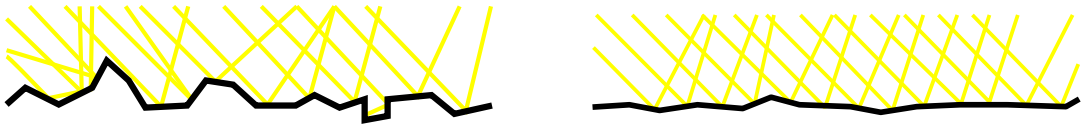
$$L = \frac{\mathrm{d}^2 \Phi}{\mathrm{d}\, A\, \mathrm{d}\, \omega \cos\theta} \tag{2.1}$$

Žiarivý tok (radiant flux) $\Phi[W]$ — udáva energiu, ktorá je vyžiarená, pohltená, prenesená alebo odrazená za jednotku času.

Ožiarenosť (irradiance) $E[W \cdot m^{-2}]$ — udáva výkon dopadajúci na plochu, resp. plošnú hustotu svetelného toku.

2.2 Fyzikálne založené vykresľovanie

Fyzikálne založené vykresľovanie (Physically based rendering; PBR) je kolekcia techník, ktoré sú odvodené od toho ako sa svetlo správa v realite. Preto výsledné osvetlenie vyzerá krajšie, resp. realistickejšie, než pri použití iných metód, napr. blinn-phong osvetľovací model. Avšak, stále sa jedná len o napodobeninu, resp. simuláciu šírenia svetla a počítačová grafika generovaná touto technikou nie je fyzikálne korektná. Napriek tomu sa v praxi používa, pretože jej výpočet je relatívne lacný a dizajnéri už nemusia používať rôzne okľuky



(a) Drsný povrch

(b) Hladký povrch

Obr. 2.3: Porovnanie povrchov modelovaných pomocou teórie mikroplôšiek. Pri drsnom povrchu sú svetelné lúče stochasticky odrazené všetkými smermi. Naopak pri hladkých povrchoch sú lúče odrazené uniformným smerom.

na to, aby svetlo v ich produkte vyzeralo správne. Taktiež, PBR unifikuje materiály, resp. sú popísané zopár konštantami, čo má za dôsledok, že za všetkých podmienok bude svetlo, ktoré s daným materiálom interaguje, vypočítané správne.

Na to, aby sme mohli určitý model nazvať fyzikálne založený, musí spĺňať 3 podmienky [39]:

1. Musí byť založený na teórii mikroplôšiek.
2. Musí spĺňať zákon o zachovaní energie.
3. Musí používať fyzikálne založenú BRDF.

2.2.1 Model mikroplôšiek

Všetky PBR modely musia byť založené na teórii mikroplôšiek. Mikroplôšky si možno predstaviť ako mikroskopické zrkadlá na povrchu objektu. Drsnosť objektu je potom daná tým aký veľký je rozptyl natočenia týchto zrkadiel (obr. 2.3). Drsnosť taktiež súvisí s tým, ako sa daný objekt bude javiť. Pokiaľ je rozptyl plôšiek veľký, bude sa objekt javiť ako difúzny, teda štatisticky bude svetlo odrážať všetkými smermi rovnako. Naopak, pokiaľ budú plôšky natočené jedným smerom, objekt bude dokonalé zrkadlo.

V skutočnosti nie je povrch žiadneho objektu dokonale hladký a preto tento model považujeme ako fyzikálne založený. Avšak, tento model dobre aproximuje realitu a pokiaľ sú plôšky dostatočne malé pri vykresľovaní, dokážeme ich modelovať pomocou *roughness* parametru.

2.2.2 Bidirectional reflectance distribution function - BRDF

BRDF je funkcia, ktorá má ako vstupné parametre:

ω_i — uhol dopadajúceho svetla, resp. smer od svetla k danému bodu povrchu

ω_o — uhol odrazeného svetla, resp. smer ku kamere

n — normála

a — parameter, udávajúci vlastnosti povrchu (typicky drsnosť povrchu)

Táto funkcia vráti ako výstup aproximáciu pomeru svetla, ktoré je odrazené smerom ω_o na základe materiálových vlastností povrchu objektu. Napr. pokiaľ by bol materiál daného povrchu dokonalé zrkadlo, BRDF by vrátila hodnotu 1 len v smere $\omega_i = \omega_o$. Vo všetkých iných smeroch, by funkcia vrátila hodnotu 0.

Ako jednoduchý príklad BRDF funkcie je phongov, resp. blinn-phongov osvetlovací model. Avšak, tento model nie je považovaný za PBR model, pretože nezachováva zákon o zachovaní energie, hoci je možné ho upraviť tak, aby zákon o zachovaní energie dodržiaval ako popisujú autori článku [27].

Väčšina rendererov v dnešnej dobe používa tzv. Cook-Torrance model [6]. Tento model počíta spekulárnu a difúznu zložku zároveň a tieto medzi sebou lineárne interpoluje (rovnica 2.2) a tým docieli zachovanie energie.

$$f = k_d \cdot f_{lambert} + k_s \cdot f_{CookTorrance} \tag{2.2}$$

$$f_{lambert} = \frac{color}{\pi} \tag{2.3}$$

$$f_{CookTorrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} \tag{2.4}$$

Parametre DFG sú v poradí distribučná funkcia normál, Fresnelova funkcia a geometrická funkcia. Typicky sa renderery líšia v použití rôznych funkcií. V tejto práci boli použité funkcie popísané v [19].

Distribučná funkcia normál

Táto funkcia aproximuje rozptyl natočenia mikroplôšiek na základe drsnosti materiálu. Disneyho GGX/Trowbridge-Reitz distribučná funkcia má dobrý pomer v kvalite výstupu voči náročnosti výpočtu (rovnica 2.5), kde $\alpha = roughness^2$, n je normála a h je vektor medzi svetlom a kamerou.

$$D(h, n, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2} \tag{2.5}$$

Geometrická funkcia

Geometrická funkcia štatisticky aproximuje zatienenie mikroplôšiek inými mikroplôškami (obr. 2.4). V rovnici 2.6 je popísaná úprava Schlickovho modelu [36], ktorá lepšie aproximuje realitu [19].

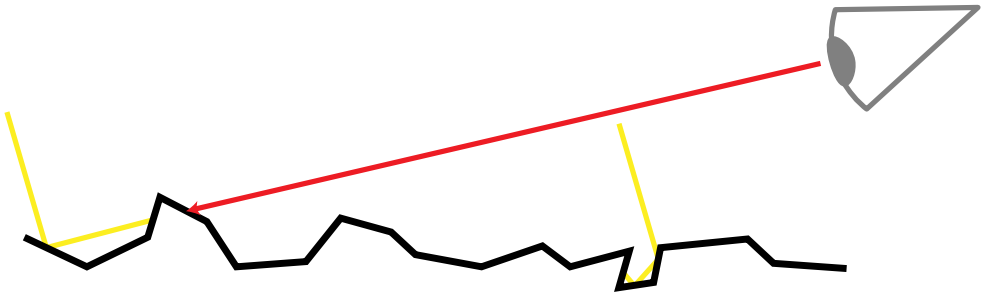
$$G(l, v, h) = G_1(l) \cdot G_1(v) \tag{2.6}$$

$$G_1(v) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k} \tag{2.7}$$

$$k = \frac{(roughness + 1)^2}{8} \tag{2.8}$$

Fresnelova funkcia

Táto funkcia popisuje pomer odrazeného svetla voči zalomenému svetlu pri dopade na materiál. Fresnelov vzťah hovorí, že pokiaľ sa pozeráme na objekt od tzv. *kritického uhlu*, lom sa mení na odraz. Každý materiál má určitú odrazivosť F_0, ktorá sa spočíta z indexu lomu materiálu (**IOR**).



Obr. 2.4: Na obrázku vznikajú dva typy zatienenia mikroplôšiek – **vľavo**, kedy je svetelný lúč zatienený inou mikroplôško a **vpravo**, kde je svetelný lúč zatemnený, resp. pohltený geometriou.

V práci bola vybraná Schlickova aproximácia [36] fresnelovho vzťahu (rovnica 2.9).

$$F_{Schlick}(h, v, F_0) = F_0 + (1 - F_0)(1 - (h \cdot v))^5 \tag{2.9}$$

2.2.3 Vykresľovacia rovnica

Vykresľovacia rovnica je momentálne najlepší model na simulovanie svetla v počítačovej grafike. Vykresľovacia rovnica slúži na výpočet výslednej žiarivosti v bode x zo všetkých okolitých smerov [16].

$$L_o(x, \omega_o, t) = L_e(x, \omega_o, t) + \int_\Omega f_r(x, \omega_i, \omega_o, t) L_i(x, \omega_i, t)(\omega_i \cdot n)\, \mathrm{d}\, \omega_i \tag{2.10}$$

$L_o(x, \omega_o, t)$ — výsledná žiarivosť bodu x v smere ω_o

$L_e(x, \omega_o, t)$ — svetlo vyžarované bodom x

$\int_\Omega$ — integrál nad pologuľov

$f_r(x, \omega_i, \omega_o, t)$ — BRDF

$L_i(x, \omega_i, t)$ — svetlo dopadajúce na bod x v smere ω_i

$(\omega_i \cdot n)$ — útlm svetla v závislosti na uhlu dopadu

Avšak, analytické riešenie tejto rovnice neexistuje a pri počítaní globálnej iluminácie sa všetky osvetlené body stávajú zdrojmi svetla a naivný iteratívny výpočet nie je možný v reálnom čase. Z týchto dôvodov vznikli rôzne metódy, ktoré sa snažia aproximovať túto rovnicu na úkor časovej či priestorovej zložitosti, resp. výslednej kvality obrazu.

Kapitola 3

Metódy globálnej iluminácie

V tejto kapitole sú predstavené používané techniky na riešenie globálnej iluminácie v reálnom čase. Hoci Raytracing, resp. pathtracing nie je technika, ktorá generuje výsledný obraz v reálnom čase, je tu uvedená, pretože sa používa ako tzv. „ground truth“, resp. na porovnanie kvality výsledku iných metód, pretože generuje veľmi kvalitné až realistické výsledky pri dostatočnom počte vyslaných lúčov.

3.1 Ray tracing/Path tracing

Táto metóda je motivovaná myšlienkou ako sa správa svetlo v prírode. Z každého zdroja svetla sú vrhnuté lúče, ktoré sa odrážajú v scéne, až narazia na priemetňu kamery. Toto sa však veľmi zle mapuje na dnešný hardvér, pretože počet vyžarovaných lúčov zo svetla je nekonečný a zároveň nie všetky lúče dopadnú do kamery.

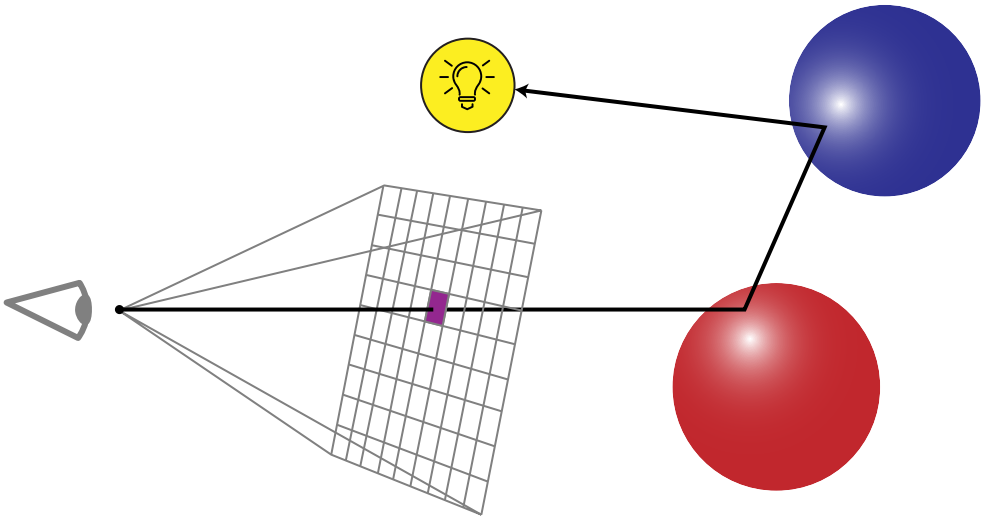
Preto sa typicky vrhajú lúče z kamery, a nie zo svetla. Lúče vyslané z kamery sa nazývajú primárne. Po zasiahnutí objektu v scéne, sa podľa jeho materiálových vlastností lúč odrazí alebo zalomí, prípadne oboje a vznikne sekundárny lúč. Avšak, pred tým, ako sa bude pokračovať v ceste, vyšle sa ďalší, tzv. „shadow“ lúč, smerom k svetlu (obr. 3.1). Tento lúč sa odráža v scéne až pokiaľ sa neuspokojí podmienka na jeho ukončenie, ako napr. počet odrazov, alebo ukončenie ruskou ruletou, v ktorej sa štatisticky ukončujú cesty po určitom počte krokov, resp. keď už daný lúč nebude mať väčšie svetelné prírastky [35].

Pri náraze vyslaného lúča na difúzny povrch vzniká „nekonečne“ veľa sekundárnych lúčov. Takýto spôsob by nebol výpočetne upočítateľný a musí sa pristupovať k iným technikám. Jedna z nich je technika nazývaná *Monte Carlo Integration* [35]. Štatisticky sa vyberie len jedna cesta, v ktorej sa bude pokračovať. Týmto spôsobom je avšak nutné vyslať mnoho primárnych lúčov, aby výsledný obraz nebol zašumený, kvôli stochastickému výberu prvkov.

Taktiež, typicky scéna obsahuje množstvo svetiel, z ktorých sa štatisticky vždy vyberie len jedno pomocou techniky nazývanej *multiple importance sampling*. Výber svetla, prípadne konkrétneho bodu z plošného svetla, môže značne znížiť počet vzoriek potrebných na výsledný obraz [38].

3.2 Radiosity

Rádiozita [12] bola metóda navrhnutá prvotne pre simuláciu tepelného žiarenia, avšak neskôr prevzatá v počítačovej grafike na výpočet osvetlenia, ktoré je taktiež forma žiarenia.



Obr. 3.1: Na obrázku je zobrazený *ray tracing*, resp. *path tracing*, ktorý možno chápať ako podmnožinu *ray tracingu*. Vyslaný lúč sa odráža od objektov v scéne až narazí na svetelný zdroj.

Táto metóda vychádza z princípu zachovania energie, v ktorej sa scéna delí na elementárne plôšky, na ktoré dopadá svetlo. A teda pre zachovanie energie, je suma žiarivého toku jednotlivých plôšiek rovná žiarivosti daných svetelných žiaričov. Taktiež, po dopade svetla na plôšku sa táto plôška stáva svetelným žiaričom. Z tohoto plynie, že táto metóda je inkrementálna, aby bola použiteľná v reálnom čase.

$$B(x,t) = E(x,t) + \rho(x,t) \int_{\Omega} B(\omega_i, t) F(x, \omega_i, t)(\omega_i \cdot n)\, \mathrm{d}\, \omega_i \tag{3.1}$$

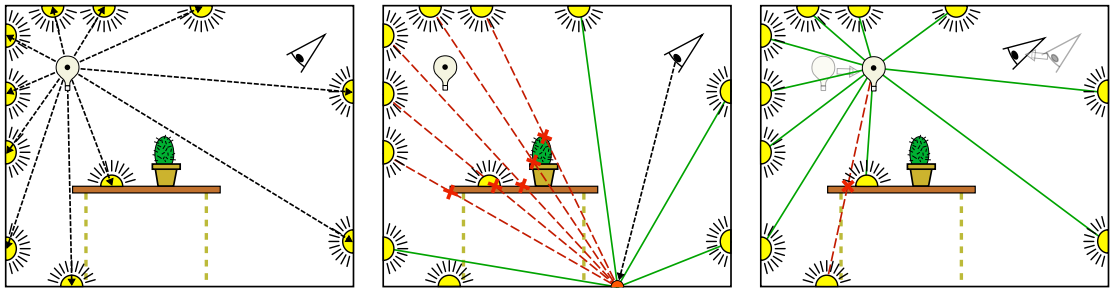
Rovnica rádiozity 3.1 je podobná s vykresľovacou rovnicou 2.10 s tým rozdielom, že metóda rádiozity predpokladá dokonale difúzne (odráža svetlo každým smerom rovnako) povrchy a teda chýba v nej smerový vektor ω_o. Preto nie je možné touto metódou vypočítať spekulárne odrazy alebo priehľadnosť.

3.2.1 Instant radiosity

Na základnú metódu rádiozity nadväzujú pokročilejšie metódy ako *instantná rádiozita* [21], alebo *inkrementálna instantná rádiozita* [24].

Metóda instantná rádiozita, alebo taktiež nazývaná *virtual point lights* (VPL), je metóda ktorá sa skôr podobá metóde *ray tracingu* než rádiozite. V instantnej rádiozite sa lúče vysielajú zo svetelných zdrojov a v mieste ich dopadu sú vytvorené VPL, ktoré možno chápať ako miniatúrne bodové svetlá rozmiestnené v scéne, pomocou ktorých je následne vykreslený finálny snímok s tieňmi a globálnou iluináciou.

Inkrementálna instantná rádiozita nadväzuje na túto metódu a snaží sa optimalizovať znovupoužitie vytvorených VPL a udržanie dobrej distribúcie v scéne aj pri pohyblivých svetelných zdrojoch (obr. 3.2). Týmto spôsobom metóda dosahuje lepšie výsledky, pretože znovupoužitím VPL sa nemusia každý snímok vykresľovať *shadow mapy* všetkých VPL, ale len určitej množine, čo zabezpečí, že je možné použiť až stovky VPL v reálnom čase.



(a) (b) (c)

Obr. 3.2: Princíp *inkrementálne instantnej rádiozity*, obrázok prevzatý z [24]. (a) Pri klasickej metóde rádiozity sa každý snímok vytvoria jednotlivé VPL. (b) Následne pri vykreslovaní fragmentu sa testuje viditeľnosť jednotlivých VPL voči danému osvetľovanému fragmentu. (c) Pri metóde *inrekemtnálnej instantnej rádiozity* sa pred vykresľovaním každého snímku testuje, ktoré VPL sa dajú znovu použiť, resp. ktoré VPL už nie sú viac validné.

3.3 Photon mapping

Photon mapping (mapovanie fotónov) [14, 15], ako už názov napovedá, je metóda, v ktorej sa počíta šírenie fotónov v scéne. Táto metóda je dvoj-priechodová. V prvom priechode sa tak ako pri *path tracingu*, stochasticky sledujú lúče zo svetelných zdrojov, ktoré sa šíria v scéne podľa materiálových vlastností objektov, na ktoré dopadajú. Po dopade každého fotónu na objekt, sa jeho pozícia a farba zaznamenáva vo fotónovej mape. V druhom priechode sa následne vykresľuje geometria a počíta globálna iluminácia pomocou tejto fotónovej mapy.

Typicky má photon mapping veľkú náročnosť na pamäť, pretože všetky fotóny sa musia uložiť do 3D textúry. Taktiež na prehľadávanie tejto mapy musí byť použitá nejaká akceleračná štruktúra, napr. *k-d* strom. Tak ako aj ostatné metódy globálnej iluminácie, aj photon mapping produkuje šum, avšak na nízkych frekvenciách, ktoré sú ťažšie zachytiteľné okom (na rozdiel od *monte-carlo* metód, ktoré generujú vysoko frekvenčný šum). Obecne pri tejto metóde platí, že čím viac vrhnutých fotónov v scéne bude, tým kvalitnejší (správnejší) výsledný snímok bude.

3.4 Voxel cone tracing

Voxel cone tracing [9] je pokročilá metóda globálnej iluminácie, ktorá dokáže simulovať dva odrazy svetla. Táto metóda je založená na voxelizácii scény, ktorá je akcelerovaná uložením do tzv. *sparse voxel octree*, v ktorej dáta – jednotlivé voxely, obsahujú žiaru v danom diskrétnom „bode“ scény. Voxelizácia je implementovaná tak, že sa scéna vykreslí trikrát – vždy pozdĺž inej osy s rozlíšením voxelov na najnižšej úrovni stromu. Dáta tohoto stromu nie sú typicky uložené priamo v akceleračnej štruktúre, ale v 3D textúre tzv. *memory pool* ako *bricks* (tehly). Pre statické scény stačí voxelizáciu vypočítať len raz, avšak pri dynamických sa musí dogenerovať, prípadne vypočítať celá od začiatku.

Po vytvorení tohoto stromu sú jednotlivé svetlá, resp. žiara propagovaná z listov tohoto stromu do jeho vyšších úrovní, typicky rasterizáciou scény z pohľadu každého svetla. V poslednom kroku je scéna vykreslená z pohľadu kamery tak, že pre každý fragment sa spočíta

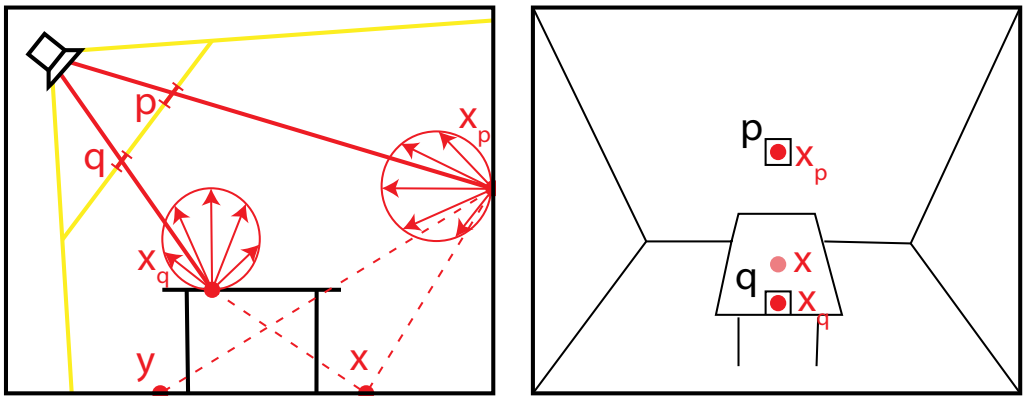
priame osvetlenie zo svetiel a nepriame pomocou tzv. difúznych kuželov, ktorými sa zisťuje, ktoré vyššie úrovne zo stromu sa majú použiť.

3.5 Reflective shadow maps

Metóda *reflective shadow maps* [10] (RSM) vychádza z metódy *shadow mapping*. V metóde *shadow mapping* sa scéna vykresľuje z každého svetla a jednotlivé texely textúry reprezentujú hĺbkovú mapu, resp. pri výpočte tieňov slúžia na test, či je daný fragment osvetlený alebo nie.

RSM generuje okrem hĺbkovej mapy ďalšiu sadu textúr, ktoré pripomínajú štruktúru *G-buffer*. Každý texel z tejto sady textúr reprezentuje jedno VPL, ktoré je použité pri výpočte globálnej iluminácie ako nepriamy zdroj svetla v scéne. Pre zachovanie interaktívnych snímkov za sekundu je počet svetelných zdrojov redukovaný na stovky a jednotlivé svetlá sú vyberané štatisticky na základe *importance sampling* algoritmu.

Táto metóda avšak nerieši viditeľnosť jednotlivých VPL, čo spôsobuje tzv. *light bleeding*. Teda prenikanie svetla za geometriu aj tam, kde by nemalo byť prítomné (obr. 3.3).



(a) (b)

Obr. 3.3: Princíp metódy RSM, obrázok prevzatý z [10]. Jednotlivé VPL x_p a x_q zodpovedajú texelom q a p v reflektívnych shadow mapách. Z obrázku je taktiež vidieť, že osvetľovaný bod x je osvetlený aj virtuálnym bodovým svetlom x_q, ktorý by ho avšak nemal osvetľovať.

3.6 Light propagation volumes

Metóda *light propagation volumes* (LPV) [17] vychádza z myšlienky, že scéna je pokrytá 3D mriežkou, v ktorej sa propaguje svetlo. Tento spôsob môže pripomínať metódu *voxel con tracing* (sekcia 3.4), avšak metóda LPV využíva na propagovanie sférické harmonické funkcie (SH). Táto mriežka je implementovaná ako dvojica mriežok, kde prvá obsahuje intenzitu nepriameho osvetlenia a druhá aproximáciu geometrie (GV). Každá bunka v týchto mriežkach je inicializovaná každý snímok od základu a na jej inicializáciu je typicky použitá metóda RSM (sekcia 3.5) generujúca VPL, ktoré sú následne premenené na koeficienty SH.

Po inicializácii nasleduje fáza propagácie svetla. Propagácia je vykonaná iteratívne, kde sa v každej iterácii pre každú bunku mriežky svetlo rozšíri do jej šiestich susedných buniek.

Pri tejto propagácii svetla je taktiež vyhodnotené potenciálne zatienenie geometrie, na ktoré je použitá druhá GV mriežka.

Po dokončení vopred definovaného počtu iterácií je táto mriežka použitá na výpočet nepriameho osvetlenia pri vykresľovaní finálneho snímku. Táto metóda avšak taktiež trpí pretekaním svetla (light bleeding), ktoré je spôsobené veľkosťou LPV bunky.

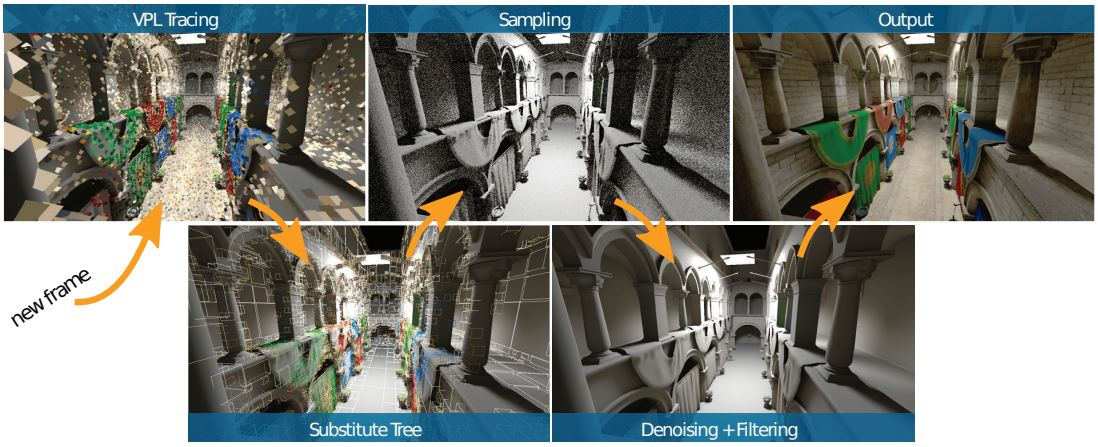
3.7 Stochastic Substitute Trees for Real-Time Global Illumination

Pri typickom výpočte osvetlenia bodu v scéne sa musí zistiť, ktorými svetlami je daný bod osvetlený, čo vedie na lineárnu časovú zložitosť. Z tohoto dôvodu metóda *lightcuts* [41] delí svetlá do zhlukov, ktoré následne ukladá v stromovej akceleračnej štruktúre. Jednotlivé svetlá sú uložené v listoch stromu, zatiaľ čo zhluky sú v jeho uzloch. Každý zhluk aproximuje výslednú intenzitu, pozíciu a materiálové vlastnosti svetla, ktoré by vzniklo zlúčením všetkých svetiel v zhluku.

Následne pri výpočte osvetlenia pre daný bod x, sa vytvorí tzv. *lightcut* – rez týmto stromom, ktorý vytvorí množinu svetiel, ktorá bude použitá na výpočet osvetlenia. Táto množina pozostáva zo zhlukov a teda redukuje počet svetelných zdrojov pre výpočet osvetlenia. Jednotlivé rezy sa každý snímok adaptívne „zjemňujú“ na základe chybovej heuristiky. Zjednodušene táto metóda prináša sublineárnu zložitosť miesto lineárnej. Avšak z toho plynie, že pre väčší počet svetiel nie je stále možné spočítať výsledné osvetlenie v reálnom čase.

Metóda *Stochastic Substitute Trees* [37] využíva hardvérovú podporu raytracingu a hlbokých neurónových sietí a je inšpirovaná predchádzajúcou metódou *light cuts*. Táto metóda v každom snímku vygeneruje určitý počet VPL, ktorý je dostatočne veľký na to, aby správne zachytil rozloženie svetiel v scéne a zároveň z nich bolo možné vytvoriť stochastický strom svetiel v reálnom čase.

Stochastický strom svetiel je akceleračná štruktúra, ktorá tak ako predchádzajúca metóda ukladá informácie o svetlách resp. jednotlivých zhlukoch. Avšak na rozdiel od predchádzajúcej metódy tento strom sa tvorí stochasticky pomocou techniky *multiple importance sampling* svetiel a zároveň sú jednotlivé zhluky uložené ako pravdepodobnostné distribúcie danej množiny svetiel.



Obr. 3.4: Zobrazenie metódy *Stochastic Substitute Trees*, obrázok prevzatý z [37].

Pri vykresľovaní sa následne tento strom prechádza pokiaľ sa nenarazí na listové VPL, alebo vhodný vnútorný uzol, resp. zhluk svetiel. Vypočítané osvetlenie použitím tohoto stromu je však zašumené kvôli stachastickému vzorkovaniu. Autori tejto metódy na odšumenie a temporálnu stabilitu použili hlbokú neurónovú sieť [3].

Táto metóda je presná aproximácia globálnej iluminácie a zároveň nespôsobuje *lightbleeding*. Avšak použitá neurónová sieť má vysoký (až 17ms) dopad na výkon.

3.8 Dynamic diffuse global illumination with raytraced irradiance fields

Tak ako aj pri ostatných metódach sa pri výpočte globálnej iluminácie musí scéna určitým spôsobom nakvantovať. V predchádzajúcej metóde [29] toto nakvantovanie spočíva v umiestnení sond (*irradiance probes*) v scéne, ktoré obsahujú informáciu o žiarivosti v danom bode scény.

Výpočet tejto žiarivosti, resp. žiarivnostných polí (*irradiance fields*) sa v predchádzajúcej metóde počíta *offline* a pri vykresľovaní sa používa táto predpočítaná hodnota. Toto má za následok, že výsledná globálna iluminácia nebude korektná pri dynamických objektoch alebo svetlách a zároveň pamäťové nároky na uloženie žiarivostných polí sú značné. Typicky boli tieto sondy rozmiestnené v scéne v pravidelnej mriežke, čo často spôsobovalo, že nejaká geometria do nich zasahovala (čo spôsobovalo zatienenie) a musela byť manuálne opravená.

Tieto dôsledky sa snaží riešiť nastávajúca metóda [25], ktorá počíta difúznu globálnu iluminánciu v reálnom čase dynamicky, za použitia hardvérového *ray tracingu* (kapitola 5.6), v ktorej jednotlivé sondy môžu byť dynamicky umiestňované do scény. Naviac táto metóda dokáže počítať „nekonečne“ veľa odrazov difúznej globálnej iluminácie tým, že jednotlivé odrazy počíta iteratívne – každý snímok jeden. Avšak, nedostatkom tejto metódy je, že dokáže počítať globálnu iluminánciu len z difúznych objektov a spekulárne odlesky sú v nej ignorované, resp. nemajú žiaden vplyv na globálnu iluminánciu.

Typický vykresľovací reťazec pri použití tejto metódy možno popísať ako:

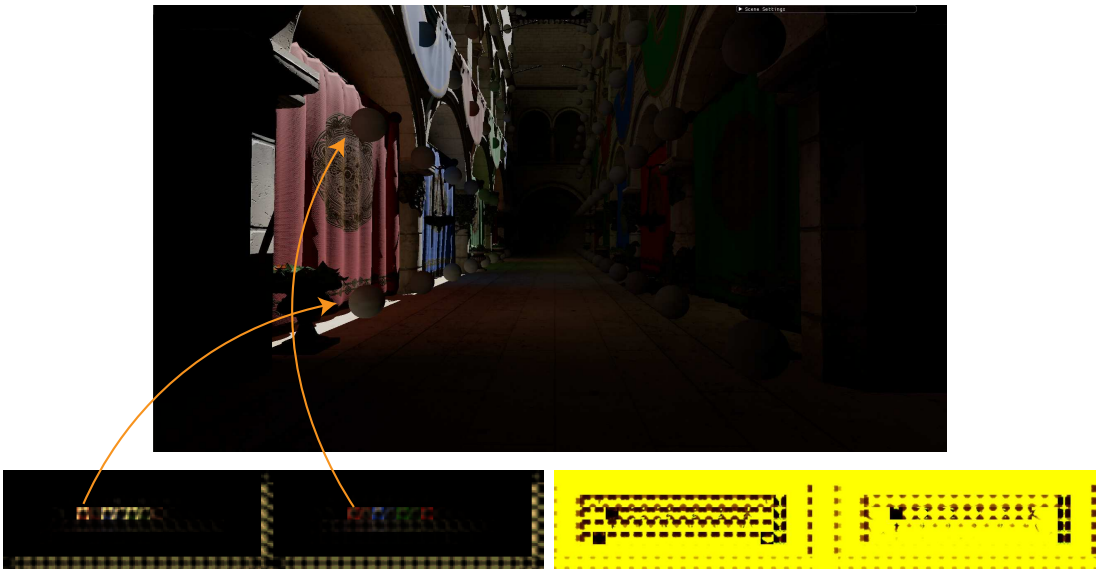
1. Sledovanie **stochastických** lúčov zo sond do štruktúry podobnej ako G-buffer.
2. Aktualizácia sond, resp. ich hĺbkových a žiarivostných máp.
3. Vykreslenie scény za použitia sond.

3.8.1 Reprezentácia sondy

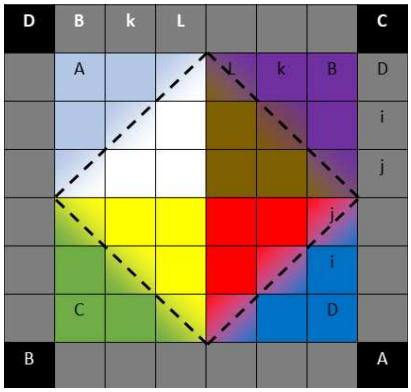
Každá sonda je reprezentovaná ako dvojica textúr, ktoré obsahujú: difúznu zložku žiarivosti a hĺbkovú mapu (obr. 3.5). Obe textúry musia mať duplikovaný okraj, kvôli správnej bilineárnej interpolácii (obr. 3.6). Mapovanie tejto 3D sondy na 2D textúru je popísané v sekcii 3.8.2. Samotný výpočet žiarivosti je uvedený nižšie v sekcii 3.8.3.

Viditeľnosť sondy

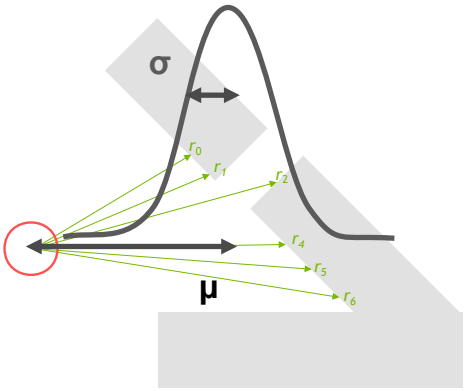
Keďže lúče zo sond sú vysielané kvázi-náhodným smerom (sekcia 3.8.3), hĺbková mapa musí byť uložená štatisticky – ako gaussovo rozloženie (obr. 3.7). Tento spôsob ďalej zabezpečí jednoduchú interpoláciu medzi predchádzajúcimi hodnotami a taktiež zmenší pamäťové nároky. Avšak je nutné podotknúť, že tento spôsob bude fungovať dostatočne dobre, len pokiaľ sú sondy rozmiestnené dostatočne husto.



Obr. 3.5: Na obrázku je zobrazená časť textúrnych atlasov sond a scéna, z ktorej boli tieto atlasy vytvorené. V ľavom atlase možno vidieť žiarivostné textúry a v pravom hĺbkové textúry. Pri hĺbkových textúrach si možno všimnúť, že sondy ktoré sú zaseknuté v geometrii, obsahujú nulovú hĺbku.



Obr. 3.6: Na obrázku je zobrazená textúra sondy, ktorá obsahuje duplikovaný okraj pre správnu bilineárnu interpoláciu. Obrázok prevzatý z [26].



Obr. 3.7: Namiesto ukladania všetkých vzdialeností lúčov je uložené len gaussovo rozloženie. Obrázok prevzatý z [28].

Do hĺbkovej textúry sondy sa ukladá vzdialenosť a vzdialenosť na druhú[1], z ktorých sa pri výpočte osvetlenia zisťuje ako ďaleko sa nachádza geometria od sondy, resp. inými slovami ako veľmi je možné dôverovať danej sonde v danom smere. Túto dôveru je možné vyjadriť pomocou Chebyshevho testu (obr. 3.7):

$$chebyshev = \frac{\sigma}{\sigma + (r - \mu)^2} \tag{3.2}$$

, kde r je vzdialenosť osvetľovaného bodu od danej sondy. Stredná hodnota a rozptyl gaussovho rozloženia sa získa ako:

$$\mu = \frac{\sum distance}{N} \tag{3.3}$$

$$\sigma = \sqrt{\frac{\sum (distance - \mu)^2}{N}} = \sqrt{\frac{\sum distance^2}{N} - \mu} \tag{3.4}$$

3.8.2 Efektívne mapovanie gule na textúru

Z článku *A Survey of Efficient Representations for Independent Unit Vectors* [5], ktorý popisuje a porovnáva množstvo komprimačných algoritmov, je algoritmus mapovania jednotkových vektorov na osemsten, resp. mapovania gule na osemsten ako najlepšia voľba v pomere veľkosti chyby a náročnosti výpočtu voči ostatným porovnávaným algoritmom.

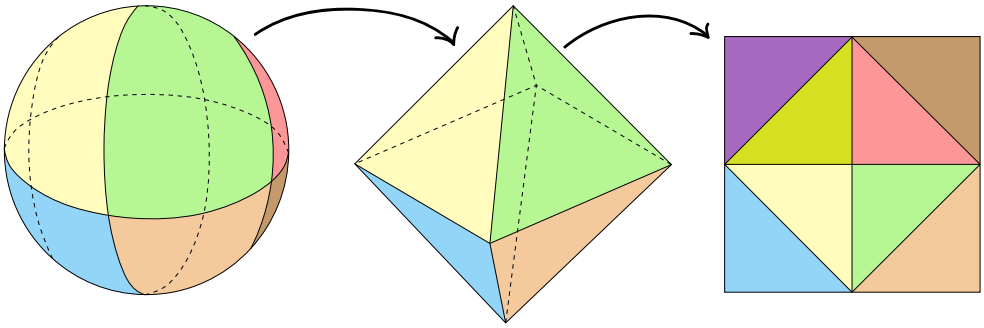
Algoritmus funguje na princípe mapovania jednotkových vektorov (ktoré majú guľový charakter) na osemsten obyčajnou zmenou definície vzdialenosti z Euklidovskej (L^2) do Manhattanskej (L^1). Následne sú jednotlivé steny osemstenu premietnuté na rovinu, v tomto prípade na 2D textúru. Postup je znázornený na obrázku 3.8. Zjednodušene povedané, jedná sa o premietnutie 3D vektoru do 2D priestoru.

Tento algoritmus sa často používa taktiež na kompresiu normál, prípadne iných vektorov.

3.8.3 Aktualizovanie sond

Aktualizácia sond prebieha v každom snímku a pozostáva z troch krokov:

[1] Vzdialenosť^2 je nutná z toho dôvodu, že počas interpolácie medzi jednotlivými iteráciami algoritmu, tieto sumy medzi sebou divergujú.



Obr. 3.8: Na obrázku je znázornené mapovanie gule na osemsten a následne premietnutie na rovinu.

1. Sledovaní n primárnych lúčov z m sond v scéne.
2. Výpočet osvetlenia v bode dopadu lúča – každý surfel textúry raytracovaných lúčov.
3. Aktualizácia textúrnych atlasov sond zmiešaním získaných hodnôt z predchádzajúcich krokov.

Sledovanie lúčov

Z každej sondy je vyslaných n primárnych lúčov, v *ray coherent* spôsobe (sekcia 5.6.2), kde n je typicky okolo 100 až 200 lúčov a tento počet je typicky odladený pre každú scénu. Lúče sa vysielajú uniformným smerom vzorkovaním sférických ekvidistantných súradníc. Tieto súradnice sú generované pomocou Fibonacciho špirály (Fibonacci lattice), ktorá je nanesená na guľu (obr. 3.9a). Naviac sú tieto lúče stochasticky otáčané pre lepšie zachytenie žiarivosti v danom bode scény (obr. 3.9b). Fibonacciho špirála sa získa pomocou sférických polárnych súradníc, kde θ je v intervale $[0, \pi]$ a ϕ je v intervale $[0, 2\pi]$:

$$\begin{aligned} x &= \sin(\theta)\cos(\phi) \\ y &= \sin(\theta)\sin(\phi) \\ z &= \cos(\theta) \end{aligned} \tag{3.5}$$

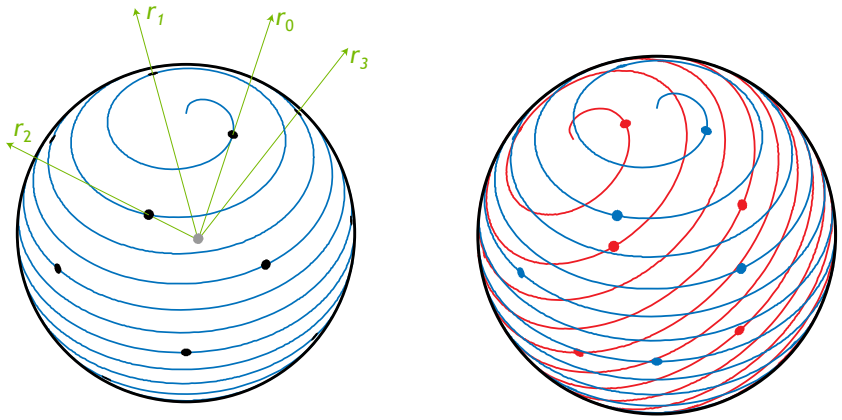
Uhly θ a ϕ sa získajú ako, kde N je počet generovaných vzoriek:

$$\begin{aligned} golden\_ratio &\approx 1.618 \\ golden\_angle &= 2\pi(2 - golden\_ratio) \end{aligned}$$

$$\begin{aligned} \theta_n &= golden\_angle * n \\ \phi_n &= \frac{\arcsin(-1 + 2n)}{N} \end{aligned} \tag{3.6}$$

Výpočet osvetlenia

Po získaní priesečníkov s geometriou sa vypočíta osvetlenie miesta dopadu tohoto lúča tou istou metódou aká bude použitá pri finálnom osvetľovaní fragmentov (sekcia 3.8.4). Pri tomto vykresľovaní sa počíta ako priame osvetlenie, tak nepriame, za použitia žiarivostí sond. Tým, že na výpočet nových žiarivostí sa používa žiarivosť sondy z predchádzajúceho snímku sa docieli to, že:



(a) (b)

Obr. 3.9: (a) Na obrázku je zobrazená fibonacciho špirála nanesená na guľu, z ktorej sú následne vysielané lúče, ktoré sú rozmiestnené uniformne. (b) Pre lepšie zachytenie žiarivosti je táto špirála každý snímok pootočená náhodným smerom.

- Výpočet globálneho osvetlenia, resp. jednotlivých odrazov, sa rozloží medzi jednotlivé snímky. Potenciálne sa takto simuluje šírenie svetla v scéne.
- Ako „vedľajší efekt“ sa vyhladí prechod medzi ostrými rádiometrickými (napr. spontánne zapnutie svetla, alebo blikanie svetla[2]) a geometrickými nespojitosťami (napr. scéna v ktorej treba množstvo n-árnych odrazov, aby osvetlenie bolo správne). Toto má avšak jeden nežiadúci efekt, že svetlo sa akoby „vlieva“ do a z oblastí s veľkou zmenou svetelného toku v danom mieste. Tento efekt je možné zmenšiť (až odstrániť) menšími hodnotami hysterézy α (popísaná nižšie), avšak nižšími hodnotami sa zhorší temporálna stabilita globálneho osvetlenia – inými slovami, scéna bude blikať.

Finálne sa tieto osvetlené lúče uložia do textúry podobnej ako *G-buffer*, pričom sa ukladá vypočítaná žiarivosť a vzdialenosť od sondy po zasiahnutú geometriu.

Aktualizácia textúrnych atlasov

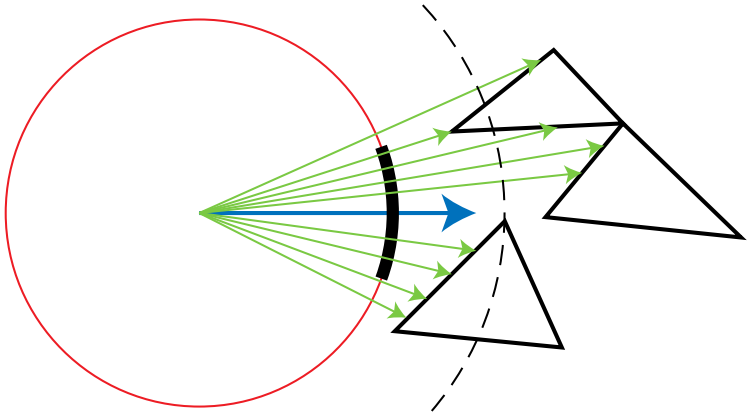
Po výpočte osvetlenia sa osvetlené surfely, ktoré sú uložené v textúre podobnej ako *G-buffer*, lineárnou interpoláciou primiešajú k texelom jednotlivých sond podľa heuristiky α. Táto heuristika udáva ako veľmi budú staré hodnoty nahradené novými (rovnica 3.7) a zaručuje temporálnu stabilitu výpočtu osvetlenia.

Ožiarenosť daného texelu sondy sa vypočíta ako suma vyslaných lúčov, resp. osvetlených surfelov (rovnica 3.8). Táto suma je normalizovaná na základe rozptylu daných lúčov (obr. 3.10), resp. kosínu vyslaného lúča ω_r a smeru daného texelu sondy ω_t. Týmto istým spôsobom sa vypočíta aj vzdialenosť texelu od geometrie.

$$new\_texel_{\omega_t} = lerp(old\_texel_{\omega_t}, radiance_{\omega_t}, \alpha) \tag{3.7}$$

$$radiance_{\omega_t} = \frac{\sum_{rays} max(0, \omega_t \cdot \omega_r) \cdot ray\_radiance}{w} \tag{3.8}$$

[2]V reálnom svete má všetko určitú zotrvačnosť a preto aj vypnutie svetla, napr. žiarovky nie je okamžité.



Obr. 3.10: Na obrázku je zobrazená 2D sonda, v ktorej je vyznačený jeden texel (hrubá čierna) s jeho smerom, a vyslané lúče zo sondy. Daný texel ukladá normalizovanú sumu vzdialeností (čiarkovaná) podľa kosínu vyslaného lúča a smeru daného texelu. Z tohoto plynie, že lúče ktoré sú bližšie k stredu texelu majú väčšiu váhu, ako lúče ktoré sú ďaleko.

Normalizačný koeficient w sa vypočíta ako:

$$w = \sum_{rays} max(0, \omega_t \cdot \omega_r) \tag{3.9}$$

3.8.4 Výpočet osvetlenia použitím sond

Výpočet osvetlenia fragmentu x je možné rozložiť do dvoch krokov, a to:

1. Výpočet priameho osvetlenia a tieňov.
2. Výpočet nepriameho osvetlenia použitím sond.

Výpočet priameho osvetlenia je možné ďalej rozložiť na priamy výpočet osvetlenia a test, či daný fragment leží v tieni, alebo je osvetlený. Osvetlenie sa teda vypočíta ako je popísané v sekcii 2.2. Na výpočet tieňov autori článku použili metódu *Variance shadow maps* [11], avšak je možné použiť akúkoľvek metódu.

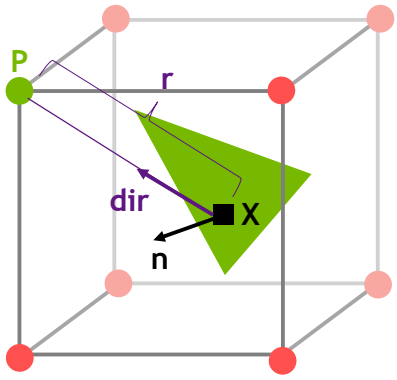
Pred výpočtom nepriameho osvetlenia sa najskôr pozícia daného fragmentu posunie podľa tzv. *normal* a *view biasu* (*self-shadow bias*), aby sa zabránilo samo-zatieneniu daného fragmentu, ktorý vzniká kvôli štatistickému odhadu vzdialenosti. Tento *self-shadow bias* sa vypočíta podľa rovnice 3.10, kde N_{bias} a V_{bias} sú nastaviteľné hodnoty.

$$normal \cdot N_{bias} + view\_direction \cdot V_{bias} \tag{3.10}$$

Pri výpočte nepriameho osvetlenia sa zistí osem sond, ktoré sa nachádzajú v okolí fragmentu x (obr. 3.11). Potom pre každú z ôsmich sond sa spočítajú váhové koeficienty a ich súčtom je finálna váha danej sondy w_i. Táto váha udáva faktor ako veľmi daná sonda prispieva k finálnemu osvetleniu. Spomenuté váhy sa spočítajú ako:

- Výpočet váhového koeficientu na vyhladenie ostrých prechodov svetla, ktoré by mohli vznikať. Tento koeficient umožní kvázi obaliť objekt svetlom, teda svetlo môže čiastočne preniknúť aj za objekt a spočíta sa ako:

$$\left(\frac{direction \cdot normal + 1}{2}\right)^2 + 0.2 \tag{3.11}$$



Obr. 3.11: Na obrázku je zobrazené vykresľovanie fragmentu x, pomocou ôsmich sond, ktoré ho obklopujú. Obrázok prevzatý z [25].

- Výpočet trilineárnej interpolácie, resp. váhového koeficientu, ktorý hovorí ako veľmi je možné danej sonde dôverovať. Čim bližšie je fragment x k sonde, tým viac bude mať daná sonda vplyv na výsledné osvetlenie fragmentu x.
- Chebyshov štatistický test na zistenie, či daná sonda môže osvetliť daný bod, resp. štatistickú aproximáciu viditeľnosti. Chebyshov test hovorí koľko prvkov nájdeme v danom rozložení v intervale 2μ. V tomto prípade, koľko geometrie je v tieni v pomere s tým, čo „videla“ sonda. Daná váha w sa vypočíta ako, kde r je vzdialenosť sondy k fragmentu x:

$$w = \begin{cases} \dfrac{\sigma^2}{\sigma^2 + (r-\mu)^2}, & \text{pre } r > \mu \\ 1, & \text{ináč} \end{cases} \tag{3.12}$$

  Taktiež Chebyshov test hovorí, že dáta musia byť v rozumnej blízkosti pri sebe [43]. A preto, aby tento test fungoval, musia byť sondy umiestnené v rozumných vzdialenostiach.

Výsledné nepriame osvetlenie sa vypočíta ako suma normalizovaných žiarivostí sond:

$$radiance = \left(\frac{\sum \sqrt{radiance_i} * w_i}{\sum w_i} \right)^2 \tag{3.13}$$

Kapitola 4

Návrh aplikácie

V nasledujúcej kapitole budú popísané len vybrané časti návrhu aplikácie, pretože výsledná aplikácia je veľmi rozsiahla. Dôraz bol kladený na výkon, teda aby vykresľovanie bolo v reálnom čase a utilizácia hardvéru čo najlepšia. Z toho vyplýva niekoľko dôsledkov, ktoré budú popísané v nasledujúcich sekciách.

4.1 Renderer interface

V čase písania tejto práce raytracing podporujú len nízko úrovňové grafické API. Z toho vyplýva, že aplikácia bude musieť byť napísaná vo *Vulkane*, alebo *DirectX 12*. Práca s týmito API môže byť veľmi náročná a preto správna dekompozícia je nutnosťou.

Renderer interface obsahuje metódy na alokáciu zdrojov, preklad shaderov, aktualizáciu zdrojov (metóda update), metóda na vykresľovanie a callbacky ako napr. `onSceneChange`, `onShaderReload`, ...

Tento interface je následne použitý pri vytváraní jednotlivých *rendererov*, ktoré implementujú spomenuté metódy. Tento návrh umožňuje dobrú dekompozíciu a jednoduchú viac vláknovú tvorbu zdrojov, preklad shaderov, vykresľovanie, atď.

Výsledná aplikácia sa skladá z nasledujúcich rendererov, ktoré budú popísané v príslušných sekciách:

Deferred renderer – Vykreslenie geometrie do G-bufferov a následné vykreslenie na obrazovku 4.2.1.

Reflections renderer – Výpočet sekundárnych až n-árnych odrazov 4.2.4.

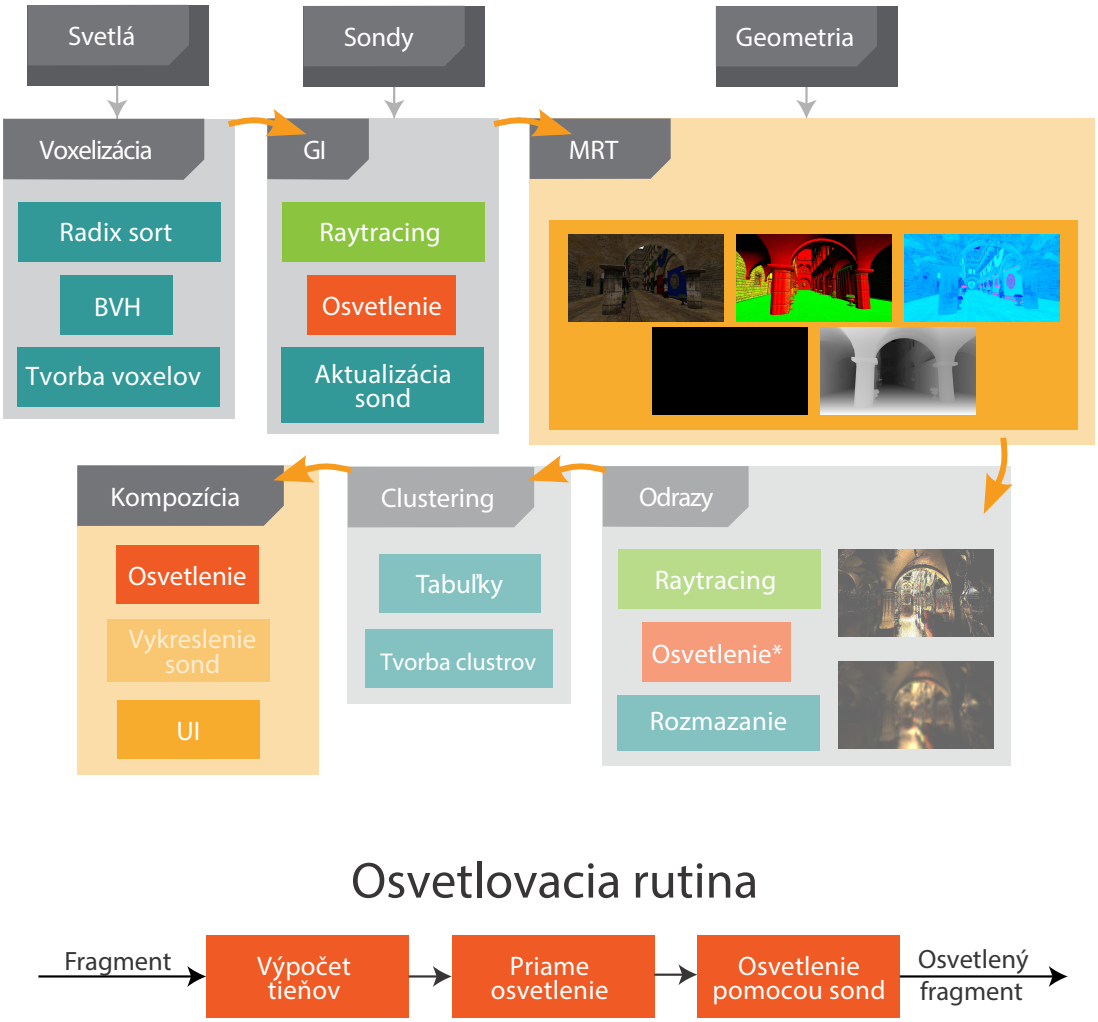
GUI renderer – Vykreslenie grafického užívateľského rozhrania (GUI) 4.2.5.

Frustum light culling renderer – Zostavenie clustrov pre clusterred shading a priradenie ich svetiel 4.3.1.

Voxel renderer – Zostavenie voxelov, priradenie svetiel a taktiež tvorba BVH stromu svetiel 4.3.2.

Global illumination renderer – Výpočet globálnej iluminácie 4.5.

Probe visualization renderer – Vykreslenie sond do scény 4.5.1.



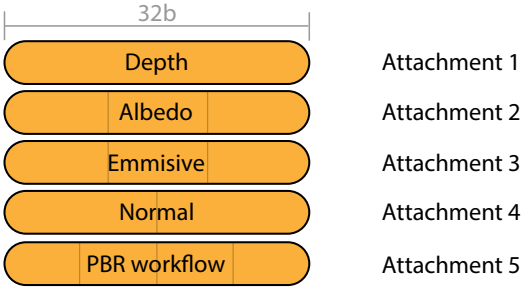
Obr. 4.1: Na obrázku je zobrazený vykresľovací reťazec aplikácie. Fáza *clustering* a *odrazy* je nepovinná a dá sa v aplikácii vypnúť. Taktiež vo fáze výpočtov odrazov je použité len nepriame osvetlenie pomocou sond.

4.2 Vykresľovací reťazec

Celý vykresľovací reťazec je zobrazený na obrázku 4.1, ktorý sa skladá z jednotlivých rendererov, ktoré implementujú vyššie spomenutý *renderer interface*.

4.2.1 Deferred renderer

Na vykresľovanie bol vybraný *deferred shading*, ktorý spočíva z vykreslenia geometrie do tzv. *G-Bufferov* (sekcia 4.2.2), ktoré sú ďalej použité na výpočet ako priameho tak nepriameho osvetlenia. Jeho obmedzenia sú väčšia náročnosť na priepustnosť pamäte a nemožnosť vykreslenia priehľadnej geometrie. Avšak tento problém sa zvyčajne rieši tým, že priehľadná geometria sa vykreslí v separátnom forward/forward+ priechode, alebo v dnešnej dobe a nástupe tzv. hybridných rendererov sa priehľadná geometria vykreslí pomocou raytracingu, kedy sa ešte naviac simulujú efekty ako lom svetla, kaustiky a iné. Avšak, navrhnutá aplikácia nepodporuje priehľadnú geometriu a pri vykresľovaní je priehľadnosť ignorovaná.



Obr. 4.2: G-Buffer pozostáva z piatich textúr (attachments), do ktorých sa kreslí: hĺbka, albedo, emisná a textúra normál, a textúra *PBR workflow*, ktorá obsahuje textúry *metallic, roughness* a *occlusion* textúru, pričom posledný kanál ostal nevyužitý.

Aby bolo možné vykresľovať viacero svetiel, je nutné svetlá vhodným spôsobom zhlukovať. Prvotne bol vybraný *clustered shading*, avšak ten nie je použiteľný pri globálnej iluminácii, každopádne ostal vo výslednej aplikácii pre možný výpočet priameho osvetlenia. Druhý navrhnutý spôsob zhlukovania je tradičná voxelizácia scény. Obe metódy využívajú akceleračnú štruktúru *Bounding Voluem Hierarchy* (BVH) na uloženie svetiel v scéne (sekcia 4.4).

4.2.2 Štruktúra G-bufferov

Štruktúru G-bufferov tvorí celkovo 5 textúr. Pri navrhovaní bol kladený dôraz na to, aby sa čo najviac znížili pamäťové nároky a zároveň sa zlepšila dátová koherencia, čo bude viesť k urýchleniu výpočtu. Vizualizácia štruktúry G-bufferu je znázornená na obrázku 4.2.

Pre hĺbku bol použitý 32b dátový typ, pretože ako možno vidieť v obrázku 4.2, pozícia fragmentu sa neuchováva, ale dopočítava sa z hĺbky a pozície kamery, čo šetrí priepustnosť pamäti. Pre farebné textúry (albedo, emission), ktoré vyžadujú tri kanály[1] je použitý dátový typ `VK_FORMAT_B10G11R11_UFLOAT_PACK32`. Tento dátový typ podporuje hardvérovú „kompresiu“, v ktorej sa jednotlivé farebné kanály ukladajú do jedného 32b *floatu* a zároveň tento dátový typ podporuje **HDR** (high dynamic range). Pre emisnú textúru by bolo avšak výhodnejšie použiť 24b dátový typ, aby sa ušetrila priepustnosť pamäti, avšak takýto typ moderné grafické karty nepodporujú z výkonnostných dôvodov.

Použitím kompresného algoritmu popísaného v sekcii 3.8.2 je pre normály použitá textúra s dvomi 16b kanálmi. Pre textúry potrebné na výpočet PBR (metallic, roughness, occlusion) bola použitá klasická textúra s 4×8b kanálmi, z ktorých posledný alfa kanál ostal nevyužitý.

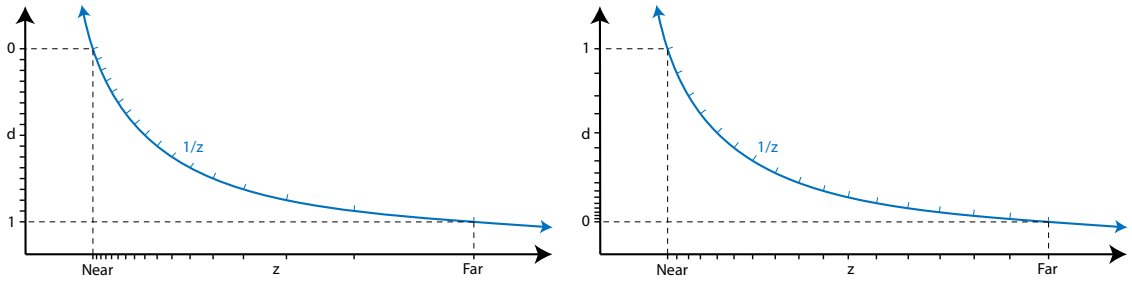
Týmto spôsobom sa podarilo efektívne skomprimovať všetky potrebné textúry do 20B na jeden fragment s tým, že nevyužitý je len 1B[2].

Reversed Z-buffer

Pri počítaní pozície z hĺbky nastáva pri vzdialených bodoch problém s presnosťou. To je spôsobené tým, že pri klasickej perspektívnej projekcii, je z hodnota mapovaná na hodnotu $0 \rightarrow 1$, kde 0 je priradená hodnote *near plane* a 1 *far plane*, podľa matice 4.1.

[1] Aplikácia nepodporuje priehľadnosť, takže alfa kanál je zanedbaný.

[2] Pokiaľ by sme rátali so spomenutým mrhaním v emisnej textúre, budú nevyužité 2B, čo je stále relatívne zanedbateľná hodnota.



(a) Krivka štandartnej z-projekcie (b) Krivka reverznej z-projekcie

Obr. 4.3: Porovnanie kriviek štandartnej a reverznej z-projekcie. (a) Pri klasickej projekcii, sú hodnoty zhlukované blízko *near plane* a postupne rednú smerom k *far plane*. (b) Pri reverznej projekcii sú hodnoty rozmiestnené kvázi-uniformne.

$$\boldsymbol{P} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & \frac{far}{near-far} & \frac{nearfar}{near-far} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.1}$$

Pri výpočte projekčnej hodnoty z_{proj} z view space hodnoty fragmentu z_{view} dostávame:

$$\boldsymbol{P} \cdot \boldsymbol{p}_{fragment} \implies z_{proj} = \frac{\frac{far}{near-far} \cdot z_{view} + \frac{nearfar}{near-far}}{-z_{view}} \tag{4.2}$$

Čo sa dá prepísať ako:

$$a\frac{1}{-z_{view}} - b \tag{4.3}$$

Z tohoto plynie, že mapovanie lineárnej hĺbky je na hyperbolickú $\frac{1}{z}$ krivku. Hodnoty blízko near plane sú mapované husto a s postupným približovaním k far plane ich hustota klesá (obr. 4.3a). Taktiež, z princípu fungovania dátového typu float je známe, že hodnoty v blízkosti 0.0 budú hustejšie, ako hodnoty v blízkosti 1.0.

Tento fakt sa dá využiť a mapovanie otočiť – hodnoty v blízkosti *near plane* budú mapované na 1 a hodnoty *far plane* na 0:

$$\boldsymbol{P}_{inv} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & -\frac{far}{near-far} - 1 & -\frac{nearfar}{near-far} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.4}$$

Potom sa táto hyperbolická nelinearita kvázi vyruší s mapovaním hodnôt v dátovom type float a dostávame semi-ekvidistančné hodnoty, pričom pri ich použití na výpočet *world-space* hodnoty daného fragmentu nedochádza k tak veľkej chybe pri väčších hodnotách z a vypočítané *world space* koordináty sú veľmi presné.

Výpočet tangent space

Na ušetrenie priepustnosti pamäte a taktiež vykrytie možných pamäťových latencií výpočtom, sa tzv. *tangent space* nepredpočítava, resp. nie je uložený v atribútoch vertexov – je

počítaný v shaderoch, keď je potrebný. Tangent space sa používa na správne mapovanie normál, pretože normály sú v ich textúrach uložené tak, že vždy smerujú smerom $+z$. *Tangent space* určuje priestor, ktorý je relatívny k povrchu daného trojuholníka [40]. Použitím špecifickej transformácie vieme následne tento priestor previesť napr. do *view space* [4]. Výpočet je naznačený v algoritme 4.1.

Avšak tento výpočet je nutné upraviť pri použití raytracingu, pretože funkcie `dFdx` a `dFdy` nie sú prístupné a je nutné ich nahradiť. Ich náhrada je popísaná v algoritme 4.2.

```
vec3 q1 = dFdx(inWorldPos);
vec3 q2 = dFdy(inWorldPos);
vec2 st1 = dFdx(inTexCoord);
vec2 st2 = dFdy(inTexCoord);

vec3 N = inNormal;
vec3 T = normalize(q1 * st2.t - q2 * st1.t);
vec3 B = -normalize(cross(N, T));
mat3 TBN = mat3(T, B, N);

vec3 normal = normalize(TBN * tangentNormal);
```

Výpis 4.1: Algoritmus transformácie normály z *tangent space* do *view space*

```
vec3 q1 = (v1.position - v0.position);
vec3 q2 = (v2.position - v0.position);
vec2 st1 = (v1.uv - v0.uv);
vec2 st2 = (v2.uv - v0.uv);

float dirCorrection = (st2.x * st1.y - st2.y * st1.x) < 0.0 ? -1.0 :
    1.0;

// Use default UV direction, if triangle has degenerative UVs
if (st1.x * st2.y == st1.y * st2.x)
{
    st1 = vec2(0.0, 1.0);
    st2 = vec2(1.0, 0.0);
}

vec3 N = vertNormal;
vec3 T = normalize(q1 * st2.t - q2 * st1.t) * dirCorrection;
vec3 B = -normalize(cross(N, T));
mat3 TBN = mat3(T, B, N);

return normalize(TBN * tangentNormal);
```

Výpis 4.2: Upravený algoritmus transformácie normály pre shader raytracingu

4.2.3 Výpočet osvetlenia

V aplikácii sa používajú dva druhy výpočtu osvetlenia – priame osvetlenie a nepriame. Pre priame osvetlenie sa používa PBR osvetľovací model popísaný v sekcii 2.2. Pre nepriame osvetlenie sa používa metóda *Dynamic diffuse global illumination with raytraced irra-diance fields* popísaná v sekcii 3.8 a sekcii 4.5. Finálne osvetlenie fragmentu je naznačené v pseudokóde 4.3.

```
1   for (light : lights)
2       if (!testShadow(fragment, light))
3           color += computeLightFlux(fragment, light);
4
5   color += getGI(fragment, light);
6   color *= ambientOcclusion;
7   color += emissiveTexture;
8
9   color = toneMap(color);
```

Výpis 4.3: Pseudokód výpočtu osvetlenia pre daný fragment

Tone mapping

Tone mapping je metóda na prevod HDR (high dynamic range) snímku do LDR (low dynamic range). V navrhnutej aplikácii je kvôli tomu, že vykreslený snímok je v HDR a typický užívateľ nebude mať monitor, čo podporuje HDR zobrazovanie. Tieto metódy ako vedľajší efekt prevodu upravia kontrast a svietivosť snímku.

V aplikácii sú navrhnuté tri metódy a je možno ich v reálnom čase prepínať pomocou GUI. Tieto metódy sú:

UE3 – metóda ktorá bola používaná v Unreal Engine 3.

Simple – jednoduchá exponenciálna funkcia.

ACES – čiastočne upravená ACES metóda, na vyzdvihnutie tmavých farieb.

Výpočet tieňov

Na výpočet tieňov sa v dnešnej dobe používa množstvo metód, z ktorých azda najpoužívanejšia je *shadow mapping* a jej varianty. Táto metóda je veľmi efektívna do určitého počtu dynamických svetiel, avšak má niekoľko zlých vlastností.

Jednou z nich je, že pre každé svetlo sa musí vykresliť separátna *shadow mapa*, v dostatočnom rozlíšení. Tento spôsob je veľmi náročný na priepustnosť pamäte a dokonca aj pri 4K *shadow mapách* nemusí byť ich rozlíšenie dostatočné a vzniká shadow aliasing (zubaté tiene). Taktiež množstvo pixelov v týchto mapách môže ostať nevyužitých.

Aj z týchto dôvodov bol vybraný raytracing na výpočet tieňov. Výpočet tieňov potom prebieha tak, že z každého svetla, ktoré potenciálne zasahuje fragment, sa vyšle lúč s nastavenou maximálnou dĺžkou smerom k fragmentu a pokiaľ tento lúč zasiahne nejakú geometriu, znamená, že daný fragment je zatienený. Ak nie, pre daný fragment prebehne výpočet priameho osvetlenia.

4.2.4 Spekulárne a glossy odrazy

Na výpočet spekulárnych a glossy odrazov je azda najpoužívanejšia metóda tzv. *screen-space reflections* (SSR), v ktorej sa pomocou *depth* buffera a pozície kamery dopočíta odraz v scéne. Toto má avšak dôsledok, že pokiaľ vypočítaný lúč zasiahne oblasť mimo *view-space* (objekty mimo zorného poľa, skryté za inými objektami, alebo objekty za kamerou, atď.), nie je informácia o odraze dostupná a musí sa pristúpiť k iným technikám, ktorou najjednoduchšia je, že sa výsledná hodnota lúča nastaví na preddefinovanú hodnotu.

Použitím raytracingu sa týmto problémom dá vyhnúť a zároveň je možné počítať odrazy na zrkadlových plochách[3]. Sledované lúče sa vysielajú na polovičnom rozlíšení na ušetrenie výpočtového výkonu a taktiež priepustnosti pamäte.

Keďže v tejto úrovni vykreslovacieho reťazca (obr. 4.1) je geometria už vykreslená do G-bufferov, nie je nutné sledovať prvotný lúč z kamery. Pozícia dopadu a smer odrazeného lúča sa dopočíta z hĺbkovej mapy a pozície kamery. Odrazený lúč sa vždy vysiela pod uhlom dopadu, pričom drsnosť povrchu, resp. rozptyl sa zanedbáva a je neskôr aproximovaný rozmazaním. Lúč v mieste dopadu je osvetlený pomocou informácie zo sond použitých pri výpočte globálnej iluminácie (sekcia 4.5). Týmto spôsobom sa nemusí vysielať n ďalších lúčov na výpočet n-árnych odrazov a tento efekt je simulovaný jednotlivými odrazmi globálnej iluminácie, čo má však za dôsledok, že n-árne odrazy nebudú spekulárne, ale len difúzne.

Následne sú osvetlené lúče zapísané do dvoch textúr ktoré obsahujú hodnotu žiare a pomerovú **blur** hodnotu rozmazania daného texelu. Táto hodnota je vypočítaná na základe drsnosti povrchu, z ktorého bol lúč vyslaný a vzdialenosti povrchu a kamery.

Rozmazanie prebieha v separátnom compute shadery, v ktorom sa počíta *gaussovský blur* [32]. Tento algoritmus je separabilný a teda je možné ho počítať v dvoch priechodoch, pričom v každom priechode sa rozmaže iná dimenzia. Po druhom priechode sa finálna textúra upraví na základe **blur** hodnoty – rozmazaná textúra a originálna textúra sa lineárne zinterpolujú podľa tejto hodnoty. Pre fragmenty, ktoré sú blízko kamery a zároveň povrch, na ktorý dopadol prvotný lúč nie je drsný, ostávajú hodnoty nerozmazané.

4.2.5 GUI

Pri vývoji sa často stáva, že GUI systém ovplyvňuje iné systémy aplikácie (napr. po stlačení tlačítka, GUI aktualizuje stav iného systému). Tento spôsob často vedie na množstvo chýb, ktoré sa veľmi ťažko hľadajú a následne ladia.

Preto bol tento systém navrhnutý tak, aby nikdy neinteragoval s inými systémami, resp. aby vždy menil len svoj stav. K tomuto používa tzv. *kontext* – dynamická mapa, ktorá obsahuje pár *kľúč, hodnota*, kde kľúč je textový reťazec a hodnota môže byť akýkoľvek dátový typ.

Pokiaľ dôjde k nejakej udalosti, napr. stlačenie tlačítka, GUI len aktualizuje svoj *kontext* a teda nemusí mať žiadne informácie o existencii iných systémov. Systém, ktorý sa zaujíma o danú hodnotu, si vie túto hodnotu zistiť z *kontextu*.

[3]Typicky bude odraz zo zrkadla zasahovať objekty mimo zorného poľa a teda SSR nebude použiteľné vo väčšine prípadov.

4.3 Light culling

Aplikácia bola navrhnutá, aby umožňovala vykresliť 2^{15} bodových svetiel. Aby bolo možné vykresliť tak vysoký počet svetiel v reálnom čase, je potrebné svetlá určitým spôsobom zhlukovať, resp. obmedziť počet svetiel, ktoré zasahujú osvetlovaný fragment a teda znížiť počet výpočtov pre tento fragment. Bez zhlukovania bolo možné metódou *brute force* vykresliť len jednotky svetiel, prinajlepšom desiatky, než začal výkon drasticky padať pod 30 FPS, čo sa nedá považovať za zobrazovanie v reálnom čase. Navrhnuté boli dve techniky – clustered shading a voxelizácia scény.

Clustered shading je technika nepoužiteľná pri výpočte globálnej iluminácie, pretože globálna iluminácia sa musí počítať všade v scéne a clustered shading využíva len *view-space* informácie. Napriek tomu, táto technika ostala v aplikácii a preto v nasledujúcej sekcii je popis jej princípu veľmi stručný.

4.3.1 Clustered shading

Clustered shading [34] je metóda na vykresľovanie väčšieho počtu svetiel v reálnom čase a vychádza z predchádzajúcej metódy tiled shading [33]. Obe metódy sa snažia zredukovať počet svetelných kalkulácií a hlavne znížiť nároky na priepustnosť pamäte, čo predstavuje najväčší problém metódy deferred shading, pretože pre každý fragment zasiahnutý svetlom musia byť prečítané informácie z G-bufferov. Optimalizácia týchto metód spočíva v radení svetiel do *screen-space tiles* (dlaždíc) alebo *clustrov* pri metóde clustered shading. Výpočet osvetlenia potom prebieha len nad listom svetiel zasahujúcich danú *tile*, resp. *cluster* [18].

Clustered shading optimalizuje predchádzajúcu metódu tým, že eliminuje hĺbkovú diskontinuitu, ktorá v nej vzniká. Tvorba jednotlivých clustrov spočíva v rozdelení pohľadového ihlanu (*view frustum*) na jednotlivé clustre. Pre optimálnosť algoritmu sa view frustum pozdĺž z-osi delí exponenciálne[4] [18, 34]. Celý algoritmus možno popísať v nasledujúcich krokoch:

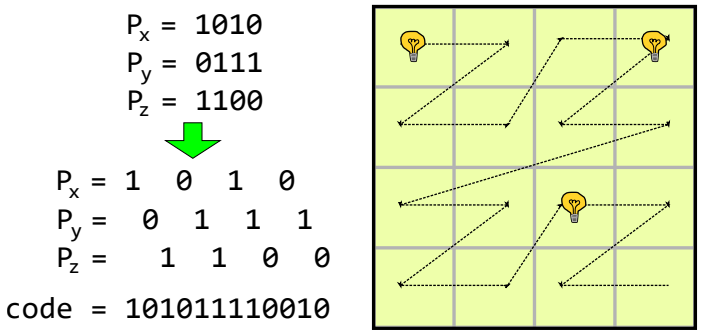
1. Priradenie fragmentov do clustrov
2. Vytvorenie unikátnych clustrov
3. Zoradenie svetiel podľa Mortonovich kódov
4. Vytvorenie BVH stromu svetiel
5. Priradenie svetiel do clustrov

4.3.2 Voxelizácia scény

Scéna sa voxelizuje pravidelným rozdelením na AABB (axis-aligned bounding box), ktorým sú následne priradené svetlá. Algoritmus voxelizácie scény možno rozdeliť do nasledujúcich krokov:

1. Priradenie Mortonoveho kódu svetlám
2. Zoradenie svetiel na základe Mortonových kódov

[4] Pokiaľ by sa view frustum delilo lineárne, kvôli perspektívnej projekcii by vzniknuté clustre blízko kamery boli sploštené a naopak vzdialené clustre by boli roztiahnuté.



Obr. 4.4: Znázornenie Mortonovej krivky pre pozíciu svetiel v dvoch dimenziách. Mortonova krivka mapuje súradnice bodu do jednej dimenzie postupným prekladaním jednotlivých bitov súradníc daného bodu.

3. Vytvorenie BVH stromu svetiel
4. Priradenie svetiel voxelom

Mortonov kód

Mortonov kód sa vytvorí postupným prekladaním jednotlivých bitov pozície daného svetla (obr. 4.4). Pozícia svetiel musí byť prevedená na celočíselný dátový typ a následne orezaná na hodnotu ±512. Toto orezanie vyplýva z toho, že Mortonov kód obsahuje 32b, z ktorých horné 2 bity sú nevyužité a 3×10b ostáva na jednotlivé dimenzie.

Z tohoto vyplýva, že svetlá môžu mať rozsah len od ±512 jednotiek a vzdialenejšie svetlá budú posunuté na túto hranicu, čo môže značne degradovať vzniknutý BVH strom. Toto sa dá riešiť tým, že dané pozície svetiel budú normalizované na pozíciu kamery, alebo ich pozície škálované, prípadne oboje zároveň. V navrhnutej aplikácii však ani jedna z týchto úprav nie je implementovaná, pretože testovacie scény nie sú rozsiahle. Vytvorenie Mortonovho kódu je znázornené v pseudokóde 4.4 [20].

```
function expandBits(vector)
{
    vector = (vector * 0x00010001) & 0xFF0000FF
    vector = (vector * 0x00000101) & 0x0F00F00F
    vector = (vector * 0x00000011) & 0xC30C30C3
    vector = (vector * 0x00000005) & 0x49249249
    return vector
}

function morton3D(position)
{
    // clip to 10 bits
    position = min(max(position + 512.0f, 0.0f), 1023.0f)
    position = expandBits(position)
    return (position.x << 2) + (position.y << 1) + position.z
}
```

Výpis 4.4: Pseudokód tvorby mortonovho kódu

4.3.3 Zoradenie svetiel

Na zoradenie svetiel bol navrhnutý paralelný radix sort optimalizovaný pre grafické karty podľa [1]. Radix sort je stabilný radiaci algoritmus, ktorý postupne radí jednotlivé kľúče po častiach. Navrhnutý radix sort je **štvorcestný**, teda radí 8 bitov (256 bins) naraz a zoradí jednotlivé kľúče po štyroch krokoch, ktoré budú ďalej nazývané **priechody**. Každá pracovná skupina (*thread block*) radí N kľúčov a táto množina kľúčov sa bude ďalej nazývať **dlaždica** (*tile*). Algoritmus typického neoptimalizovaného radix sortu možno popísať v nasledujúcich krokoch:

1. **Upsweep** – vytvorenie lokálnych histogramov číslic pre každú dlaždicu
2. **Scan** – vytvorenie prefixovej sumy všetkých číslic za použitia vypočítaných histogramov
3. **Downsweep** – zoradenie číslic a zápis do globálnej pamäte

Pri takomto postupe vzniká $3N$ prístupov do pamäti a tieto tri fázy sa musia počítať pre každý jeden **priechod**. V článku *A Faster Radix Sort Implementation* [1] sú fázy upsweep a scan spojené do jednej tzv. **onesweep** fázy a autori optimalizovali túto fázu na $2N$ prístupov do pamäti. Celý algoritmus radenia je naznačený na obrázku 4.5.

Onesweep

V tejto fáze sa najskôr vytvorí tzv. **histogram prepass**, čo je globálny histogram kľúčov pre každý **priechod** (čiže pri navrhnutom štvorcestnom priechode, sa vytvoria štyri histogramy, každý o veľkosti 256 bins). Používa sa fakt, že aj po zoradení bude tento histogram pre každý priechod nezmenený a teda stačí ho vytvoriť len raz, na začiatku radenia. Tento histogram je použitý na získanie počiatočnej pozície jednotlivých kľúčov v danom priechode (obr. 4.5).

Nasleduje fáza decoupled look-back, ktorá počíta prefixovú sumu a využíva štruktúru **partícia**. Táto štruktúra sídli v globálnej pamäti a má veľkosť $tiles \times bins$, čo je vlastne množina lokálnych histogramov pre každú *tile* s meta-informáciami. **Bin** je 32b integer (obr. 4.6), ktorého spodných 30b určuje počet a vrchné 2 bity sú určené pre metódu decoupled look-back a určujú značky:

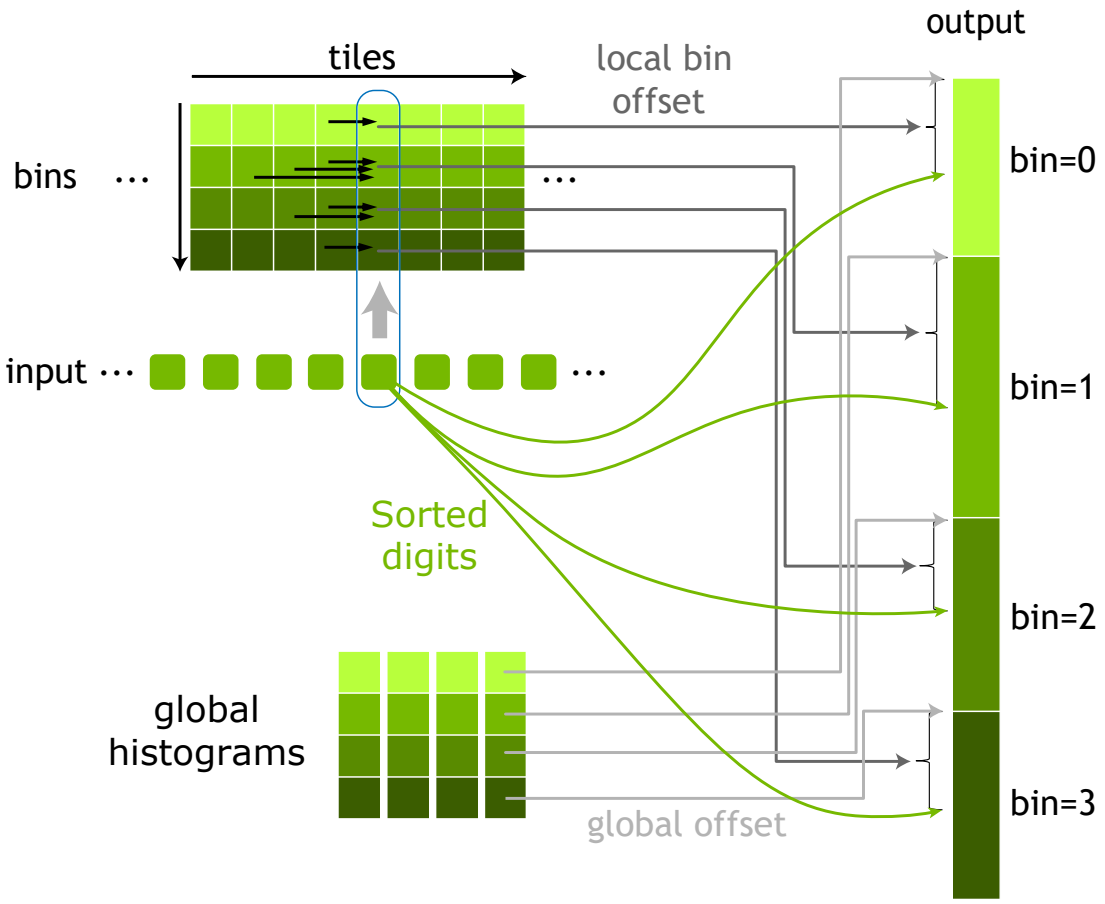
A – Agregát (hodnota) pre daný bin bola zapísaná.

P – Inkluzívna prefixová suma pre daný bin je aktuálna.

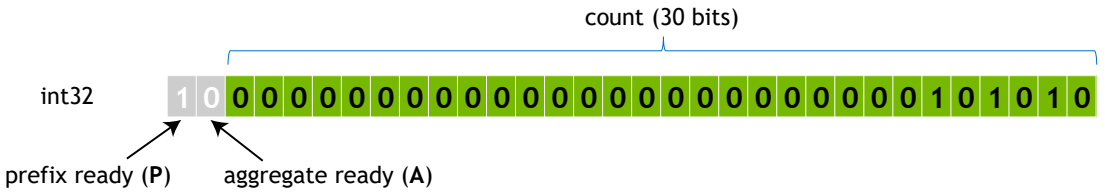
X – Indikuje, že daný bin nebol ešte inicializovaný.

Metóda *Decoupled look-back* umožňuje paralelný výpočet prefixovej sumy v jednom priechode [31] a pracuje nasledovne:

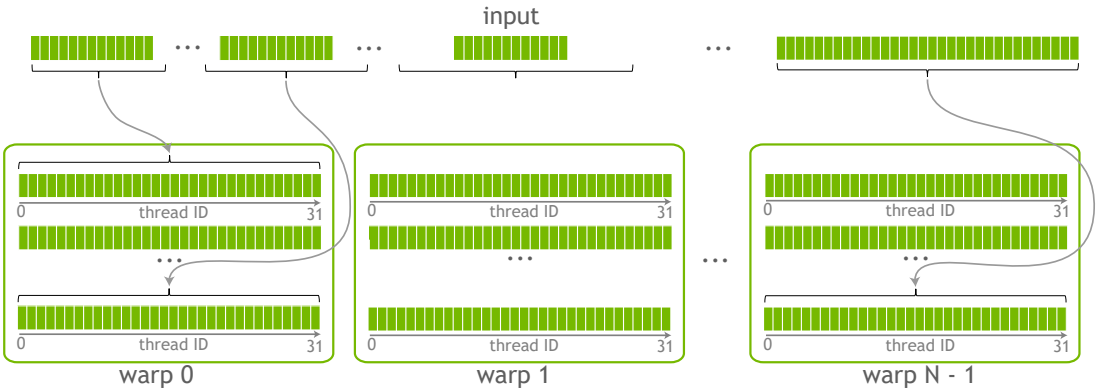
1. **Inicializácia partície a synchronizácia** – nastavenie statusových značiek v štruktúre partícia na hodnotu **X** a synchronizácia všetkých pracovných skupín.
2. **Spočítanie lokálneho agregátu** – všetky pracovné skupiny spočítajú lokálny histogram pomocou *atomicAdd* a zapíšu ho do partície s tým, že aktualizujú statusovú značku na **A**. Pokiaľ sa jedná o pracovnú skupina číslo 0, táto nastaví svoju značku na **P** a končí, pretože jej prefixová suma je 0.



Obr. 4.5: Na obrázku je zobrazený celý algoritmus optimalizovaného paralelného radix sortu, inšpirované z [1].



Obr. 4.6: Zobrazenie štruktúry *Bin*. Jedná sa o 32b celočíselný typ, ktorého dva najvrchnejšie bity obsahujú statusové značky. Obrázok prevzatý z [1].



Obr. 4.7: Poradie kľúčov pri radení, obrázok inšpirovaný z [1].

3. **Exkluzívna prefixová suma** – Každá pracovná skupina počíta svoju prefixovú sumu atomickým čítaním statusových značiek, počínajúc predchádzajúcou dlaždicou $(n-1)$.

   Pokiaľ je jej statusová značka nastavená na hodnotu:

   X – pokračuj krokom 3.

   A – hodnota dlaždice, resp. jednotlivých *bins* sledovanej dlaždice je pripočítaná do interného registru **exkluzívnej** prefixovej sumy pracovnej skupiny a pokračuje sa krokom číslo 3 s tým, že „sledovaná“ dlaždica bude „pred-predchodca“ $(n-2)$.

   P – predchádzajúca dlaždica obsahuje aktuálnu **inkluzívnu** prefixovú sumu. Táto suma sa pripočíta k internému registru **exkluzívnej** prefixovej sumy a pokračuje sa nasledujúcim krokom.

4. **Zápis inkluzívnej prefixovej sumy** – Pracovná skupina zapíše svoju **inkluzívnu** prefixovú sumu do **partície** a následne aktualizuje svoju značku na hodnotu **P**. **Inkluzívna** prefixová suma sa spočíta ako suma agregátu a **exkluzívnej** prefixovej sumy interného registra.

Upsweep

V tejto fáze sa zoradia a zapíšu kľúče do globálnej pamäte. Okrem toho sa však znova počíta lokálny histogram[5] s tým, že kľúče si udržia stabilný *rank* (miesto, na ktoré sa zapíše zoradený kľúč). V navrhnutom algoritme sa počíta lokálny histogram v každom *warpe* zvlášť. Toto umožňuje bezbariérové radenie v zdieľanej pamäti a teda je veľmi efektívne.

Každý *warp* obsahuje svoj lokálny histogram (256 bins) a naviac pole o 256 hodnotách, tzv. *match mask*, ktoré taktiež sídlia v zdieľanej pamäti. Aby sa zachovala stabilita tohto algoritmu, každý *warp* radí N prvkov, ktoré nasledujú tesne za sebou. Toto poradie je zobrazené na obrázku 4.7.

Následne, každý *warp* počíta svoj lokálny histogram a prefixovú sumu. Tento výpočet je znázornený v algoritme 4.5 a taktiež obrázku 4.8. Pole *match mask* slúži na hlasovanie, kde každé vlákno vo *warpe* ohlasuje, že jeho kľúč spadá do daného *bin* v histograme. Následne sa získa *leader* (vlákno s najväčším indexom pre daný bin), ktoré bude atomicky aktualizovať

[5]Prvotný histogram sa počíta rýchlo bez stability, aby sa mohli nastaviť statusové značky v metóde decoupled look-back na hodnotu **A** a teda vykryli sa latencie výpočtom.

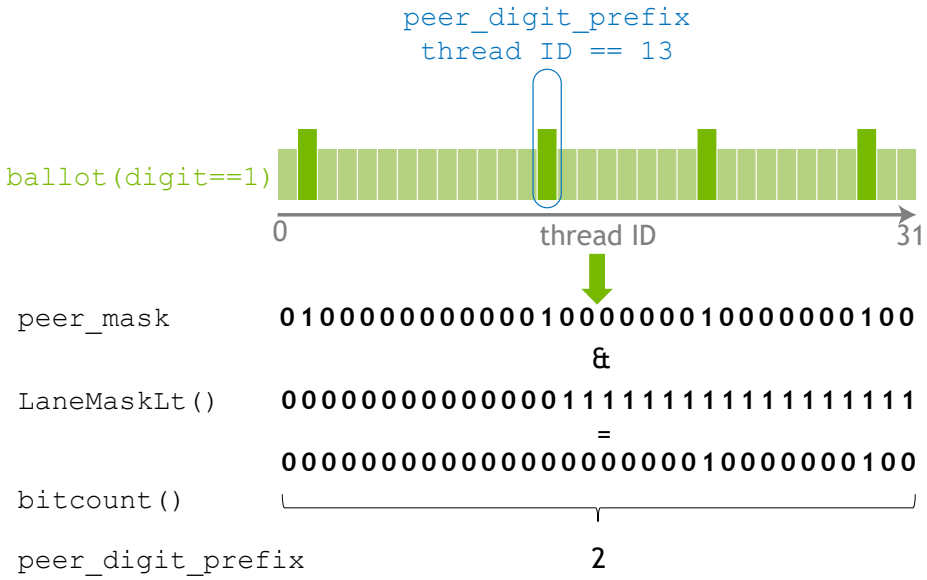
histogram. Po aktualizácii histogramu si vlákna medzi sebou vymenia počiatočnú pozíciu pre daný kľúč v histograme a zapíšu *warp-lokálny rank* pre daný kľúč.

```
// reset histogram and match mask
for (i = 0; i < bins; ++i)
{
    histogram[warpID][i] = 0
    matchMask[warpID][i] = 0
}

laneMask = 1 << threadID
laneMaskLe = laneMask | (laneMask - 1)

for (i = 0; i < lightsPerWarp; ++i)
{
    // unique key for each thread
    binIndex = getBinID(keys[i])
    warpOffset = 0

    atomicOr(matchMask[warpID][binIndex], laneMask)
    leader = findMSB(matchMask[warpID][binIndex])

    // prefix sum
    popc = bitCount(matchMask[warpID][binIndex] & laneMaskLe)

    if (leader == threadID)
    {
        warpOffset = atomicAdd(histogram[warpID][binIndex], popc)
        matchMask[warpID][binIndex] = 0
    }

    // broadcast offset from leader
    // to other threads with same bindIndex
    warpOffset = subgroupShuffle(warpOffset, leader)

    // write rank of the sorted key
    ranks[i] = warpOffset + popc - 1
}
```

Výpis 4.5: Pseudokód warp-lokálneho radenia

Po dokončení warp-lokálneho histogramu sa všetky vlákna v pracovnej skupine zosynchronizujú a vypočítajú prefixovú sumu pracovnej skupiny z týchto histogramov. Výsledný *rank* sa získa ako:

```
rank = warpPrefixSum + threadblockPrefixSum + globalPrefixSum
```



Obr. 4.8: Na obrázku je znázornený výpočet *warp*-lokálnej prefixovej sumy pre vlákno číslo 13, obrázok inšpirovaný z [1]. Vlákna vo *warpe* zistia pomocou funkcie *ballot*, ktoré majú rovnakú hodnotu (digit==1), ktorá sa uloží do premennej *peer_mask*. Pomocou bitovej masky sa následne zistí prefixová suma.

4.3.4 Priradenie svetiel voxelom

Voxely sú uchované v pamäti v dvoj-úrovňovej hierarchii (obr 4.9). Prvá úroveň predstavuje dané voxely, ktoré sa odkazujú do druhej úrovne. Druhá úroveň obsahuje stránky, do ktorých sa následne zapisujú indexy svetiel, ktoré zasahujú voxel. Tieto stránky sú alokované so špecifickou granularitou, čo má za následok potenciálne menej prístupov do pamäte a zároveň umožní indexovať väčší počet svetiel, resp. voxelov[6]. Každý voxel je 32b celočíselná hodnota, ktorá je rozdelená na dve časti:

Spodných M bitov – počet svetiel, ktorý daný voxel obsahuje.

Vrchných N bitov – index stránky, v ktorej sa nachádzajú indexy do poľa svetiel.

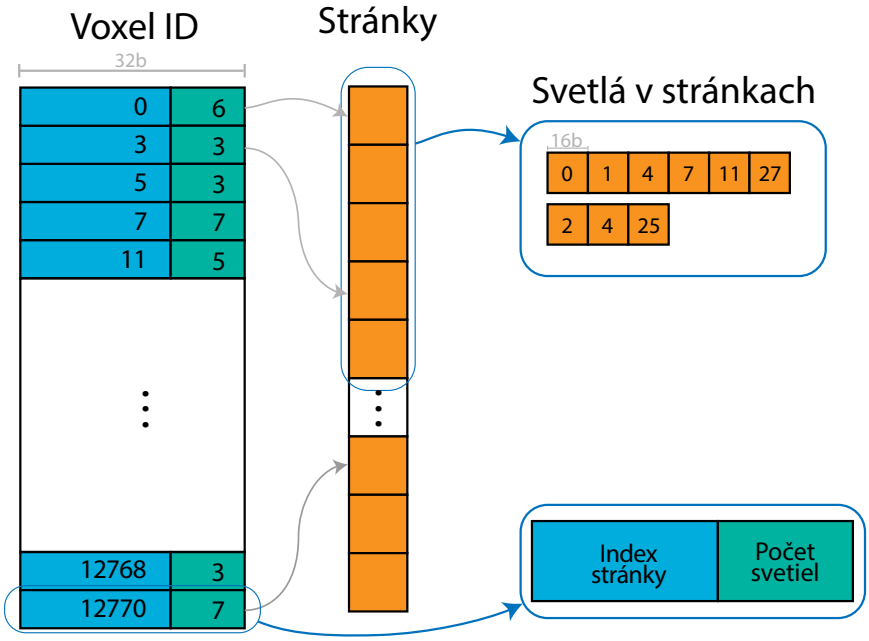
Svetlá sa priraďujú voxelom na základe priechodu BVH stromom svetiel. Jeho zostavenie a priechod je popísaný v sekcii 4.4. Pokiaľ voxel nezasahujú žiadne svetlá, je do prvej úrovne zapísaná hodnota 0 a žiadna stránka nie je alokovaná. Toto je efektívne, pretože obyčajným porovnaním na nulu sa vie zistiť, či voxel obsahuje svetlá a pokiaľ nie, nemusí sa pristupovať ďalej do pamäte, resp. pre daný fragment sa ukončí výpočet osvetlenia.

Indexovanie voxelov

Keďže voxely sú implementované ako pravidelná mriežka AABB, je ich indexovanie triviálne:

```
voxelID = (fragment_position - voxel_origin) / voxel_size
```

[6]Pokiaľ by sa ukladal len presný offset do pamäti, bolo by možné uložiť len N svetiel, pričom s použitím granularity je možné uložiť $N \cdot granularita$



Obr. 4.9: Na obrázku je znázornená dvoj-úrovňová hierarchia uloženia voxelov. Prvá úroveň predstavuje jednotlivé voxely a je bitovo rozdelená na hodnotu počtu svetiel, ktoré daný voxel zasahujú a kvázi ukazateľ na prvú stránku, v ktorej sa nachádzajú dané svetlá. Stránky v obrázku sú alokované s granularitou 2.

, kde `voxelID` je 3D vektor. Prevedenie tohoto vektora na jednorozmerný index je znázornené v algoritme 4.6. Vektor `DIMENSION` udáva počet bitov pre danú dimenziu.

```
1  index =
2      voxelID.x |
3      (voxelID.y << DIMENSION.x) |
4      (voxelID.z << (DIMENSION.x + DIMENSION.y))
```

Výpis 4.6: Prevedenie 3D indexu voxelu na 1D index

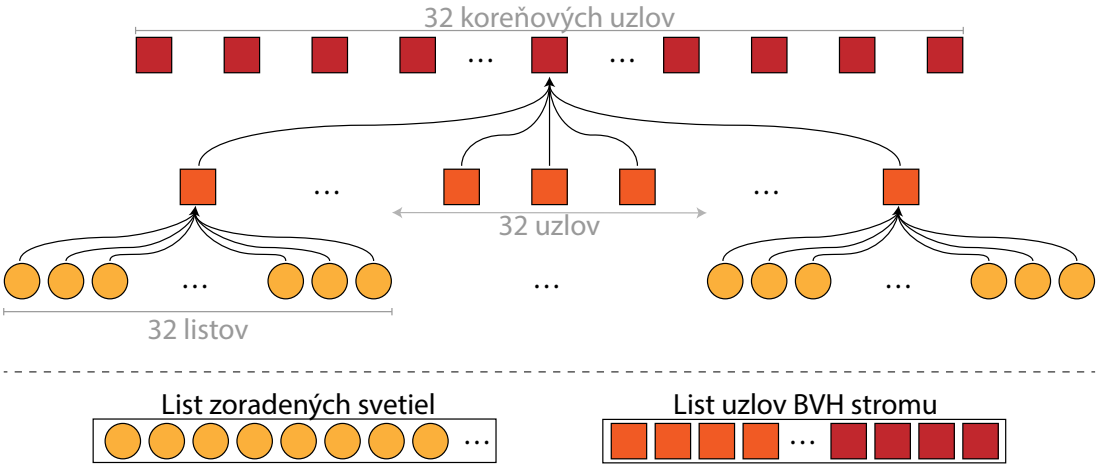
Potom prvá úroveň hierarchie bude musieť obsahovať presne

$$\texttt{DIMENSION.x} \cdot \texttt{DIMENSION.y} \cdot \texttt{DIMENSION.z}$$

prvkov, čo pri veľkosti mriežky $128 \times 128 \times 128$ predstavuje 8MB pamäte. Taktiež každému voxelu budú musieť byť priradené svetlá podľa sekcie 4.3.4, z čoho plynie, že mriežka voxelov nemôže byť príliš veľká a taktiež hustota svetiel pre danú scénu a rozmery voxelov musí byť rozumná, aby bol algoritmus vypočítateľný v reálnom čase.

4.4 BVH Strom svetiel

Bounding volume hierarchy (BVH) je akceleračná stromová štruktúra, v ktorej uzly predstavujú **axis-aligned bounding box** (AABB) a v listoch je uložená potrebná geometria (v tomto prípade konkrétne svetlá). Existuje množstvo algoritmov na budovanie BVH stromu, ktoré sa zameriavajú buď na kvalitu výsledného stromu (resp. rýchlosť jeho priechodu), alebo rýchlosť jeho stavby.



Obr. 4.10: Na obrázku je zobrazený navrhnutý BVH strom, ktorého vetviaci faktor je 32 a pri troch úrovniach dosiahne maximálny počet 2^{15} uchovaných svetiel.

V aplikácii bol navrhnutý tzv. **LBVH** (linear bounding volume hierarchy), ktorý je inherentne veľmi paralelný, avšak nezaručuje kvalitu výsledného stromu. Tento algoritmus je taktiež veľmi jednoduchý (v porovnaní napr. SAH BVH alebo HLBVH).

Navrhnutý strom je zobrazený na obrázku 4.10. Jeho vetviaci faktor je 32, čo umožňuje rýchly prechod na dnešných grafických kartách. Navrhnutá aplikácia podporuje maximálne 2^{15} svetiel. Toto množstvo sa ukázalo byť dostatočne veľké na utilizáciu navrhnutých metód a zároveň pri vetviacom faktore 32 vzniknú maximálne len tri úrovne stromu.

4.4.1 Budovanie stromu

Navrhnutý strom sa buduje zdola nahor (*bottom-up*) s tým, že listy v tomto strome „neexistujú“ – svetlá sú uložené v pamäti zoradené podľa Mortonovej krivky a index daných svetiel sa dopočíta z pozície uzlov. Jednotlivé uzly sa potom tvoria z AABB jednotlivých svetiel. Vzniknuté uzly sú uložené v pamäti tesne za sebou, čo možno vidieť na obrázku 4.10.

4.4.2 Priechod stromom

Priechod stromom je navrhnutý ako test AABB voči AABB a priechod je bezbariérový – každý *warp* prechádza stromom sám bez ohľadu na ostatné *warpy* v pracovnej skupine. Očakáva sa, že veľkosť warpu bude vždy 32. Strom sa prechádza do hĺbky a tento priechod je zobrazený v algoritme 4.7.

Princíp algoritmu spočíva v tom, že každý warp testuje jeden AABB (čo môže byť napr. voxel, alebo cluster) voči jednotlivým uzlom stromu. Pri binárnom strome je koreňoví uzol len jeden, avšak v navrhnutom LBVH strome pozostáva koreň z 32 uzlov. Takto môže každý *warp* otestovať jednotlivý uzol, či koliduje s testovaným AABB. Pokiaľ aspoň jeden warp koliduje, uloží sa bitová maska kolízií do tzv. *collision stack*, čo je zásobník kolízií pre jednotlivé úrovne stromu.

Z tohto zásobníka sa následne vždy odoberie len jeden bit (zmení sa z 1 na 0). Tento bit signalizuje kolíziu daného uzlu v danej úrovni stromu. Algoritmus sa postupne vnára do hĺbky stromu a pokračuje takýmto testovaním až narazí na listovú úroveň, kedy otestuje už priamo kolíziu svetla a testovaného AABB.

Pokiaľ zásobník kolízií obsahuje hodnotu 0 (žiaden bit nie je nastavený - kolízia neexistuje), pokračuje sa vynorením z aktuálnej úrovne. Pokiaľ je aktuálna úroveň koreňová, ukončí sa prehľadávanie.

```
currentLevel = rootLevel;

// Load root AABB node
node = createNode(levels[currentLevel].offset)

// Threads are voting if they collide with their AABB
// Bitmask is saved to collisionStack
saveBallot(currentLevel, collideAABB(node, box))

// Traversing until there are no collisions in root level
while (currentLevel <= rootLevel)
{
    // Check for collision on current Level
    if (collisionStack[warpID][currentLevel] == 0)
        currentLevel++
    else
    {
        // Save least significant bit of collision
        // And remove this bit from collisionStack
        saveLSB(currentLevel)

        // If we are at the bottom of the tree, check lights collision
        if (currentLevel == 0)
        {
            testAndWriteCollisions()
        }

        // Continue traversing tree down
        else
        {
            nodeOffset = computeNodeOffset(currentLevel);
            currentLevel -= 1;

            node = createNode(nodeOffset + levels[currentLevel].offset);
            saveBallot(currentLevel, collideAABB(node, box));
        }
    }
}
```

Výpis 4.7: Pseudokód priechodu BVH stromom

4.5 Globálna iluminácia

Globálna iluminácia bola navrhnutá tak, ako je popísaná v kapitole 3.8. Každý snímok sa *raytracuje* n lúčov z m sond. Tieto lúče sa ukladajú do textúry, v ktorej každý riadok reprezentuje jednu sondu, a každý stĺpec jeden lúč z danej sondy. Pre túto textúru bol zvolený dátový typ `VK_FORMAT_R16G16B16A16_SFLOAT`, ktorý má dostatočnú presnosť pre dané hodnoty. V `RGB` kanály je uložená hodnota osvetleného fragmentu a v poslednom kanáli je uložená vzdialenosť zásahu geometrie.

Jednotlivé lúče sú vysielané uniformne avšak stochasticky – vždy iným náhodným smerom. Pre vytvorenie tohoto náhodného smeru je použitá rotačná matica, ktorá vždy náhodným smerom pootočí vytvorenú fibonacciho špirálu. Prvotne bola táto matica vytvorená pomocou kvaterniónu, ktorý bol prevedený na rotačnú maticu. Tento kvaternión sa vytvoril vygenerovaním štyroch náhodných čísel. Pred jeho prevodom na rotačnú maticu musí byť tento kvaternión znormalizovaný, pretože kombinácia týchto náhodných štyroch čísel nemusí byť validná. Tento spôsob bol funkčný, avšak vzniknuté rotačné matice negenerovali uniformné rozloženie.

Z tohto dôvodu bola použitá zložitejšia metóda popísaná v [23]. Táto metóda vyžaduje len tri náhodné čísla a funguje v polárnom súradnicovom systéme tak, že najskôr vygeneruje náhodnú rotáciu okolo vertikálnej osi a následne otočí severný pól na náhodné miesto.

Tieto tri náhodné čísla sa prvotne vygenerujú náhodným bielym šumom a následné iterácie sú generované modrým šumom. Tento spôsob bol navrhnutý na zníženie blikania globálnej iluminácie, resp. na zrýchlenie konvergencie. Modrý šum je generovaný ako:

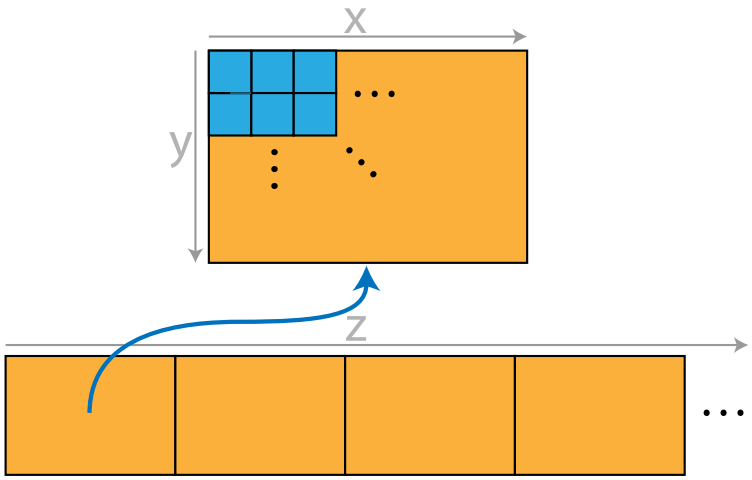
$$random_{i+1} = (random_i + golden\_ratio) \bmod 1.0 \tag{4.5}$$

Následne je textúra sledovaných lúčov použitá na aktualizáciu sond, teda primiešanie nových hodnôt na základe *alfa* hodnoty. Jednotlivé sondy sú uložené v dvoch atlasoch textúr – atlas žiare a hĺbkový atlas (atlas vzdialeností). Pre atlas žiare bola vybraná textúra s dátovým typom `VK_FORMAT_B10G11R11_UFLOAT_PACK32` a veľkosťou 6×6 a duplikovaným okrajom pre správnu bilineárnu interpoláciu (obr. 3.6). Pre hĺbkovú mapu bola vybraná textúra s dátovým typom `VK_FORMAT_R16G16_SFLOAT` a veľkosťou 14×14 a taktiež duplikovaným okrajom. Uloženie jednotlivých sond v atlase je zobrazené na obrázku 4.11.

Pre zníženie blikania scény spôsobené globálnou ilumináciou, sú vysielané lúče rozdelené na dve skupiny – *statické* a *dynamické*. Skupina statických lúčov sa vysiela vždy tým istým smerom (nie je na ne aplikovaná rotačná matica), čo spôsobí, že zmena žiare bude konzervatívnejšia a scéna osvetlená globálnou ilumináciou nebude toľko blikať, a teda môžu sa použiť nižšie hodnoty histerézy α. Dynamické lúče sú potom vždy pootočené podľa rotačnej matice spomínanej vyššie.

4.5.1 Vizualizácia sond

Na efektívne vykreslenie gulí – sond, je efektívne použitá rasterizácia. Efektívnosť spočíva v tom, že na grafickú kartu sa neprenášajú „žiadne“ dáta, resp. daný model gule je analyticky dopočítaný. Pre každú guľu sú vykreslené dva trojuholníky – *billboard* natočený ku kamere. Následne pre každý fragment tohoto *billboardu* sa analyticky spočíta priesečník lúča z kamery voči guli. Pri zásahu sa na daný fragment nanesie textúra pre konkrétnu sondu z daného atlasu. Pokiaľ daný fragment nie je zasiahnutý, diskartuje sa a na obrazovku nie je nič vykreslené.



Obr. 4.11: Na obrázku je zobrazené rozloženie textúr jednotlivých sond v textúrnom atlase. Modrý štvorec predstavuje jednotlivé sondy.

4.6 Reprezentácia scény

Na reprezentáciu scény bol zvolený textový dátový formát – **JSON**. Tento formát podporuje dátovú štruktúru, pole, objekt (dvojica kľúč-hodnota) a jednotlivé hodnoty, ktorými sú reťazce, čísla a špeciálne hodnoty *true*, *false* a *null*.

Aplikácia obsahuje jeden súbor *config.json*, ktorý v najvyššej úrovni obsahuje pole objektov – jednotlivých scén. Týmto spôsobom je potom možné prepínať medzi jednotlivými scénami za behu aplikácie. Každá scéna sa potom skladá z:

- **Mena scény**
- **Metadát globálnej iluminácie**:
  - **dimensions** – počet sond pre jednotlivé dimenzie
  - **origin** – stred okolo ktorého budú sondy rozmiestnené
  - **spacing** – odsadenie jednotlivých sond
  - **hysteresis** – predvolená hodnota hysterézy (je možné ju meniť pomocou GUI)
  - **view bias** a **normal bias** – *biasy* popísané v sekcii 4.5 (je možné ich meniť pomocou GUI)
  - **rays** – počet vysielaných lúčov pre jednu sondu
  - **static rays** – počet statických lúčov
- **Metadát statických svetiel** – tieto svetlá budú v scéne prítomné vždy a nie je možnosť ich vypnúť:
  - **type** – typ svetla:
    - **0** – priame svetlo (directional light)
    - **1** – bodové svetlo (point light)
  - **direction** – smer svetla, pokiaľ sa jedná o priame svetlo
  - **position** – pozícia svetla, pokiaľ sa jedná o bodové svetlo

  - **radius** – polomer svetla, pokiaľ sa jedná o bodové svetlo
  - **power** – itenzita svetla
  - **color** – farba svetla

- **Modelov** – predstavujú jednotlivé modely na disku. Jeden model môže mať niekoľko instancií:
  - **name** – meno modelu, ktoré slúži ako referencia pre instancie
  - **path** – cesta k modelu, ktorá môže byť buď relatívna, alebo absolútna

- **Instancie modelov** – predstavujú jednotlivé modely v scéne:
  - **object** – je referencia na konkrétny model
  - **transform** – predstavuje transformáciu daného modelu (translácia, rotácia, zväčšenie)

Kapitola 5

Implementácia

V tejto kapitole budú rozobrané implementačné detaily návrhu aplikácie popísaného v predchádzajúcej kapitole. Aplikácia je implementovaná v jazyku C++ s využitím štandartu 20 a moderných techník, ktoré prináša. Na vykresľovanie je použité Vulkan API, resp. jeho C++ wrapper, vo verzii 1.2. Preklad aplikácie je implementovaný nástrojom CMake.

5.1 Použité knižnice

Na tvorbu aplikačného okna a spracovanie vstupu z klávesnice a myši je použitá knižnica `SDL2`. Jej vybratie bolo podmienené najmä multiplatformnosťou, jednoduchosťou a taktiež tým, že je dostupná v použitom `Vulkan SDK`. Na vykresľovanie užívateľského rozhrania a nastavovanie parametrov je použitá knižnica `Dear ImGui`. Maticové, vektorové a iné výpočty zabezpečuje knižnica `GLM`.

Na testovanie a ladenie aplikácie boli použité štandardné validačné vrstvy pre Vulkan. Načítavanie modelov zabezpečuje knižnica `TinyGLTF`, ktorá interne používa knižnicu `stb_image` na načítavanie textúr. Na preklad shaderov bola použitá knižnica `shader-c`, ktorá interne používa knižnicu `glslang` a knižnice pre prácu s `SPIRV` formátom.

Pre jednoduchšiu prácu s vláknami bola použitá knižnica `OpenMP`. Taktiež pre jednoduchšiu prácu so správou grafickej pamäte je použitá knižnica `Vulkan Memory Allocator`. Táto knižnica spravuje alokáciu pamäte, pretože v nízkoúrovňových API ako je Vulkan, sa pamäť alokuje v tzv. *pool*, pričom počet týchto alokácii je obmedzený. Taktiež je možné alokovať rôzne typy pamätí určené pre rôzne účely.

Ďalej boli použité knižnice `fmt`, ktorá umožňuje veľmi efektívne formátovanie reťazcov, knižnica `Robin Hood hashing`, ktorá efektívne implementuje hashovaciu tabuľku, a ktorá je oveľa rýchlejšia než klasická `std::unordered_map`.

5.2 Načítanie scény

Načítavanie scény ja implementované pomocou knižnice interne použitej v knižnici `TinyGLTF` na parsovanie `JSON` súborov. Reprezentácia scény, resp. tohoto súboru je popísaná v sekcii 4.6. Taktiež z tohoto plynie, že aplikácia podporuje len `GLTF` modely a okrem toho, len modely bez rozšírení (čiže len modely s *PBR workflow*).

Jednotlivé modely, ktoré tvoria scénu, sú načítané viacvláknovo pomocou knižnice `OpenMP`. Model sa ďalej skladá z materiálov a primitív. Primitívum reprezentuje skupinu vertexov, resp. indexov na tieto vertexy a každému primitívu je priradený index na materiál. Načítanie

modelu predstavuje parsovanie `GLTF` súboru, vytvorenie indexového a vertexového bufferu na grafickej karte, nahratie textúr s vytvorením *mip-mapingu* a nakoniec vytvorením *BLAS* štruktúry pre raytracing.

Z tohoto plynie, že každý model má vlastný BLAS. Ideálne je vytvárať čo najmenej týchto štruktúr, pretože pri raytracovaní sa musí otestovať každá jedna štruktúra. Avšak keby sa vytvoril len jeden veľký BLAS, nebolo by možné pri zásahu geometrie zistiť aký model, resp. materiál, sa má použiť.

Materiál je štruktúra, ktorá obsahuje indexy na jednotlivé textúry použité pri vykresľovaní daného modelu. Pri vykresľovaní je použité rozšírenie `VK_EXT_descriptor_indexing`, ktoré umožňuje *bindless* indexovanie textúr. Pre správne fungovanie *bindless* indexovania, je nutné pri pristupovaní do poľa textúr použiť *intrinsic* funkciu `nonuniformEXT`.

5.3 Práca s Vulkánom

Práca s Vulkánom môže byť veľmi náročná pretože Vulkan je veľmi explicitné API. Bez žiadneho „chytrého“ úložiska sa veľmi rýchlo stane, že jednotlivé triedy sú zaplavené Vulkan objektami, čo veľmi sťažuje vývoj a zhoršuje čitateľnosť kódu. Aj z týchto dôvodov bola navrhnutá trieda, tzv. `smart resource storage`, v ktorej sú permanentne uložené všetky potrebné Vulkan objekty. V kóde 5.1 je zobrazené použitie tohoto úložiska, ktoré je veľmi často využívané v aplikácii.

Tento spôsob taktiež umožňuje, že všetky systémy v aplikácii majú prístup k Vulkan objektom, ktoré vytvorili iné systémy. Pokiaľ by však nejaký systém chcel, aby k jeho objektu nemohol pristupovať žiaden iný systém, nemusí použiť *smart storage* a objekty si vytvoriť v jeho vlastnom pamäťovom priestore, prípadne používať kombináciu oboch.

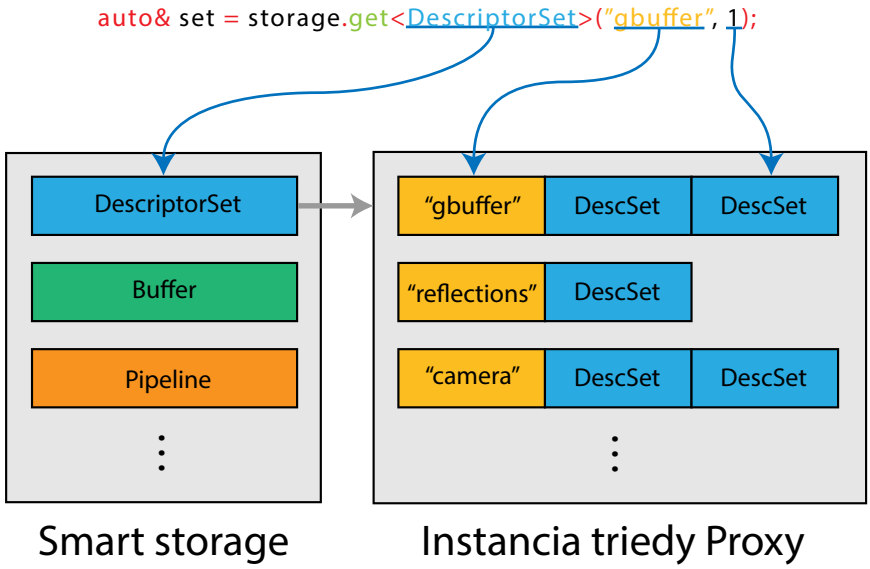
```
1  for (size_t i = 0; i < FRAMES_AHEAD; ++i)
2      mResources.add<Semaphore>("GI.probes.finished", i);
3
4  ...
5
6  auto waitSem = resources.get<Semaphore>("GI.probes.finished",
       imageIndex);
7  auto cmd = resources.get<CommandBuffer>("gbuffer", imageIndex);
```

Výpis 5.1: Kód zobrazujúci použitie chytrého úložiska Vulkan objektov.

Implementačne je táto trieda vlastne len kvázi chytré obalenie dvojúrovňovej hashovacej tabuľky, v ktorej sú pridané metódy ako `add<Proxy>(...)`, `get<Proxy>(...)`, atď. Kľúč v tejto tabuľke je `typeID` daného Vulkan objektu a hodnota je trieda odvodená od tzv. triedy `Proxy`, od ktorej sa následne odvodzujú homogénne kontajnery jednotlivých objektov, ktoré predstavujú úložisko daného objektu. Na toto homogénne úložisko je použitá taktiež hashovacia tabuľka, kde kľúč je reťazec hľadaného/pridávaného Vulkan objektu a hodnota je samotný objekt. Vyzualizácia tejto tabuľky je zobrazená na obrázku 5.1.

5.3.1 Preklad shader programov

Na preklad shaderov je použítá knižnica `shader-c`. Táto knižnica umožňuje preklad shaderov v jazyku GLSL a taktiež podporuje rozšírenie `GL_GOOGLE_include_directive`, ktoré



Obr. 5.1: Vizualizácia *Smart resource storage*. Prvá úroveň obsahuje `typeID` daného Vulkan objektu. Druhá úroveň obsahuje jednotlivé objekty.

umožňuje `#include` direktívy, teda „vkladanie“ súborov, resp. umožňuje znovupoužiteľnosť častí existujúceho kódu a jednoduchú dekompozíciu.

Naviac táto knižnica umožňuje špecifikovať definície pri preklade shaderov, ktoré boli kľúčovou súčasťou pri vývoji. Aplikácia taktiež umožňuje *online* preklad shaderov pomocou klávesovej kombinácie `shift + R`, čo umožnilo ľahší vývoj a veľmi efektívne odladovanie aplikácie.

5.3.2 Debugovanie aplíkacie

Na implementáciu bolo použité *Visual Studio* od spoločnosti *Microsoft* s použitím nástroja *Resharper* od spoločnosti *JetBrains*, ktoré spolu tvoria sadu kvalitných prostriedkov na vývoj a ladenie.

Na skúmanie a ladenie výpočtov na grafickej karte bola použitá sada programov od firmy NVIDIA: *Nsight Graphics* a *Nsight Systems*. *Nsight Graphics* je pokročilý nástroj a poskytuje možnosti na prezeranie textúr, bufferov, descriptor settov, jednotlivé drawcalls, ale aj compute dispatch a veľa iných. Tento nástroj bol zvolený najmä kvôli tomu, že v dobe začiatku písania tejto práce, podporovali raytracing len grafické karty od Nvidie a taktiež *NSight* bol jediný debugovací nástroj, ktorý umožňoval prezeranie akceleračných štruktúr a iných vecí spojených s raytracingom. Tento nástroj okrem prezerania jednotlivých grafických primitív umožňuje taktiež danú aplikáciu profilovať, skúmať výkonnosť shaderov, atď.

Na jednoduchšie ladenie debugovacích výpisov (aj z validačných vrstiev Vulkanu) bolo použité rozšírenie `VK_EXT_debug_utils`. Toto rozšírenie umožňuje pomenovať Vulkán objekty (ak dôjde k validačnej chybe u nejakého objektu, je toto pomenovanie použité vo výpise) a taktiež vložiť pomenovania pri nahrávaní *command bufferov*, ktoré sú potom zobrazené v nástrojoch ako *Nsight Graphics*, *RenderDoc*, atď.

5.4 Vykresľovanie

Typicky pri vykresľovaní v reálnom čase je potrebné každý snímok nahrať nové *command buffre*, pretože nad geometriou scény prebiehajú rôzne vyraďovacie algoritmy, prípadne sú dané objekty animované, atď. Aj napriek tomu, že v aplikácii nie je potrebné znova nahrávať *command buffre* každý snímok pretože geometria je statická, kvôli testovaniu výkonnosti a celkovo dobrým praktikám sú *command buffre* nahrávané každý snímok.

Nahrávanie *command buffrov* je výpočetne CPU náročná operácia. Aplikácia bola navrhnutá tak, aby podporovala ľubovolný počet vykresľovacích vláken, resp. vláken, ktoré nahrávajú tieto *command buffre*.

Command buffre sú alokované z tzv. `command pool`, ktorý predstavuje *thread-safe* miesto v pamäti, do ktorého sa nahrávajú jednotlivé príkazy pre grafickú kartu. Teda pri viacvláknovom vykresľovaní, treba vytvoriť N *command poolov*, kde N je počet vláken, ktoré budú nahrávať jednotlivé *command buffre*. Každému vláknu sa priradí jeden *command pool*, z ktorého si alokujú jednotlivé *command buffre* pripravené na nahrávanie.

Na abstrakciu tohoto problému je vytvorená trieda `CommandPoolManager`, ktorá implementuje vyššie spomenutú funkcionalitu a nad rámec zabezpečuje vzájomné vylúčenie pri posielaní jednotlivých *command bufferov* na frontu grafickej karty[1].

5.5 Komunikácia systémov aplikácie

Na komunikáciu jednotlivých systémov je čiastočne využívaný kontext aplikácie. Tento kontext je vlastne implementácia návrhového vzoru singleton (jedináčik), ktorý po spustený aplikácie vytvorí okno na vykresľovanie, inicializuje Vulkan, vyberie správne zariadenie atď.

V kontexte je ďalej uložený načítaný konfiguračný súbor aplikácie, resp. jednotlivých scén (sekcia 4.6), *Command buffer manager*, trieda GUI a hlavne *smart resource storage*.

Keďže je táto trieda singleton, má k nej prístup akýkoľvek systém v aplikácii. Aplikácia je navrhnutá tak, že systémy medzi sebou vlastne nekomunikujú, len zdieľajú medzi sebou Vulkan objekty, alebo menia svoj stav na základe vstupu z *GUI*.

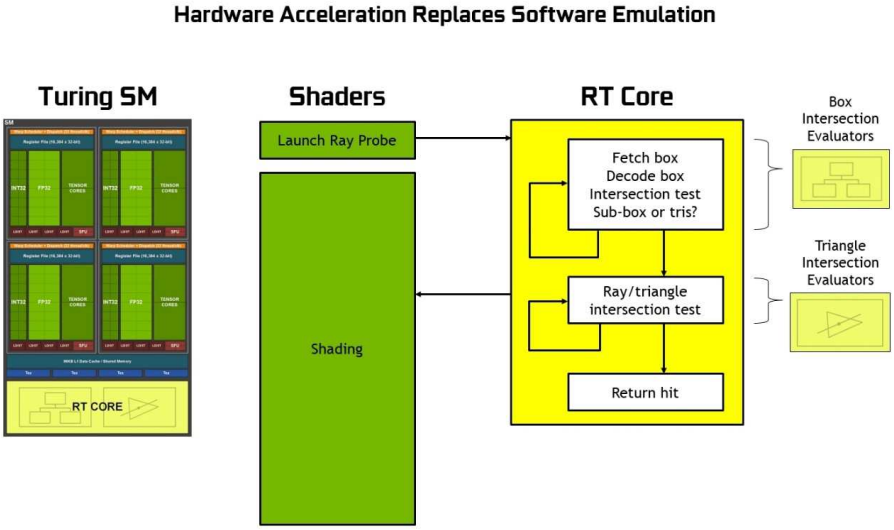
5.5.1 GUI kontext

Systém GUI využíva na uloženie stavov jednotlivých GUI prvkov *GUI kontext*. Tento kontext je implementovaný ako dvojica hashovacia tabuľka a *memory pool*. Hashovacia tabuľka obsahuje kľúč, ktorým sa vytvorí/získa daný objekt a hodnotu, ktorá obsahuje ukazateľ na pamäťové miesto v *memory pool*. Typ pamäťového miesta, resp. veľkosť alokácie je odvodená na základe templatového typu. Príklad použitia kontextu je zobrazený v algoritme 5.2.

```
1  mContext.set("probe.viewBias", scene.probes.viewBias);
2  mContext.set("probe.origin", scene.probes.origin);
3
4  ...
5
6  auto& viewBias = mContext.get<float>("probe.viewBias");
7  auto& origin = mContext.get<glm::vec3>("probe.origin");
```

Výpis 5.2: Kód zobrazujúci použitie GUI kontextu.

[1] *Submit* na grafickú kartu, resp. frontu grafickej karty môže súčasne zapisovať len jedno vlákno a explicitné vzájomné vylúčenie je nutné pri viacvláknových aplikáciách.



Obr. 5.2: Znázornenie SM jednotky a RT jadier v architektúre Turing. Obrázok prevzatý z [7].

5.6 Raytracing

V počítačovej grafike je technika rasterizácie používaná dekády, avšak pri vykresľovaní, napr. odrazov od lesklých povrchov za použitia rasterizácie je nutné použiť algoritmy, ktoré používajú zjednodušenia pri ich výpočtoch, z ktorých sa potom tvoria artefakty pri vykresľovaní finálneho snímku. S nástupom hardvérového raytracingu je možný fyzikálne správny výpočet týchto javov v reálnom čase – niečo, čo v minulosti nebolo možné.

V dnešnej praxi sa používa tzv. hybridné vykresľovanie – rasterizácia sa využíva tam, kde vyniká (na typické vykreslenie scény), tak isto raytracing (napr. na spekulárne odrazy). Typicky sa sledujú lúče na polovičnom, prípadne inom pomere rozlíšenia a obraz ktorý vznike ich sledovaním, sa následne roztiahne na výsledné rozlíšenie, napr. technológiou DLSS (*deep learning super-sampling*), ktorá využíva tenzorové jadrá a dokáže vytvoriť viac detailov vo výslednom snímku voči ostatným technikám (napr. MSAA).

5.6.1 Raytracing na grafických kartách Nvidia

Grafické karty od firmy Nvidia obsahujú od generácie *Turing* tzv. RT jadrá. Tieto jadrá akcelerujú výpočet priesečníku lúča v scéne použitím akceleračnej štruktúry BVH (bounding volume hierarchy). RT jadrá pozostávajú z dvoch špecializovaných jednotiek – jednotkou na hardvérovú akceleráciu priesečníku lúča a obalového kvádra, a druhú jednotku na akceleráciu priesečníku lúča a trojuholníka. RT jadrá pracujú samostatne, resp. asynchrónne a teda SM jednotky musia len „naštartovať“ výpočet a následne môžu vykonávať ostatné výpočty dokiaľ sa nájde priesečník lúča v scéne [7][8].

Tvorbu BVH stromu zastrešuje driver grafickej karty. BVH strom sa skladá z uzlov, ktoré obsahujú tzv. AABB (axis aligned bounding box), čiže kvádre, ktoré obklopujú geometriu ich detí. V listoch sú potom uložené už samotné primitíva – trojuholníky. Test garantuje tzv. *water-tight* spojenie trojuholníkov a teda lúč nemôže „obtekať“ medzi spojmi trojuholníkov a nemôže nastať situácia, kedy by nastalo zasiahnutie viacerých primitív v tom istom bode [42]. Celková funkčnosť RT jadier, resp. ich použitia je zobrazená na obrázku 5.2.

5.6.2 Raytracing použitím Vulkan API

Pre používanie hardvérového raytracingu poskytuje Vulkan tri rozšírenia (extensions). Tieto rozšírenia sú súčasťou jadra štandartu Vulkan 1.2 a samozrejme, je možné ich aktivovať len na kartách, ktoré podporujú hardvérový raytracing [22]:

1. `VK_KHR_acceleration_structure` — poskytuje funkcionalitu na vytváranie a aktualizovanie akceleračných štruktúr.

2. `VK_KHR_ray_tracing_pipeline` — poskytuje shader programy a pipeline využívajúce raytracing.

3. `VK_KHR_ray_query` — poskytuje tzv. *ray queries*, čo sú intrinsic funkcie na volanie RT jadier z takmer akýchkoľvek stupňov v pipeline. Avšak tento *inline* raytracing pridáva značnú záťaž na registre, čo v konečnom dôsledku vedie na nižšiu okupanciu a je vhodné použiť radšej raytracing pipeline, pokiaľ je to možné.

Akceleračné štruktúry, s ktorými následne pracuje grafická karta, sa delia na dvojstupňovú hierarchiu a to tzv. *bottom level* (BLAS) a *top level* (TLAS) akceleračné štruktúry. BLAS obsahuje trojuholníky, AABB, prípadne inú geometriu ako napr. analytický popis primitív. TLAS obsahuje referencie na BLAS, čo umožňuje instancovanie geometrie a zároveň vytvára strom, ktorý sa bude prechádzať pri hľadaní priesečníkov s geometriou v scéne. Platí, že tvorba BLAS je výpočtovo náročná operácia, zatiaľ čo tvorba TLAS je lacná operácia, avšak neodporúča sa ju zneužívať, pretože ich tvorba môže mať merateľnú záťaž [13].

Raytracing pipelines sú podobné ako *graphic pipelines* vo Vulkane, avšak poskytujú väčšiu flexibilitu. Aplikácia môže mať pre každý objekt v scéne rôzny shader a implementácia zariadi automatické prepínanie medzi programami počas vyhodnocovania priesečníku lúča so scénou. Vulkan poskytuje štyri druhy shaderov:

- **ray generation** — V tomto shadery sa generujú lúče pomocou funkcie `traceRayEXT`, ktorá akceptuje parametre ako počiatok a smer lúča (každé vlákno predstavuje jeden lúč), akceleračnú štruktúru, ...

- **any hit** — Tento shader sa zavolá pri každej potenciálnej kolízii, pretože pri hľadaní najbližšieho primitíva môže lúč naraziť do niekoľkých primitív. Taktiež je možné v tomto shadery lúč predčasne ukončiť (napr. pri zoslabení svetelného toku cez transparentné objekty).

- **closest hit** — Tento shader sa zavolá po nájdení najbližšieho primitíva, resp. jeho priesečníku.

- **miss** — Tento shader je zavolaný, pokiaľ žiaden priesečník nebol nájdený.

Pri vysielaní lúčov je potrebné dbať na koherenciu lúčov – lúče z rovnakých warpov by mali byť vyslané podobným smerom, aby dochádzalo k maximalizácii priesečníka rovnakej geometrie a teda, aby sa predchádzalo náhodným skokom v pamäti[2]. Tejto problematike, a tiež ako vylepšiť koherenciu lúčov, sa venuje článok [30].

[2]Pri zásahu geometrie sú dáta ako napr. materiálové vlastnosti, textúry, poprípade iné metadáta, uložené na inom mieste v pamäti, ako vertexy popisujúce primitíva.

Kapitola 6

Zhodnotenie a výsledky

V tejto kapitole sú popísané uskutočnené merania, zhrnutie dosiahnutých výsledkov a grafické výstupy z aplikácie pri rôznych parametroch nastavenia.

6.1 Meranie času

Aplikácia bola testovaná len na jednej zostave, ktorá je uvedená v tabuľke 6.1:

Parameter	Zostava
CPU	AMD Ryzen 7 3700X
GPU	Nvidia RTX 2060 Super
RAM	DDR4 32GB
OS	Windows 10 Pro
GPU Driver	460.89 (15.12.2020)
Vulkan SDK	1.2.162.0

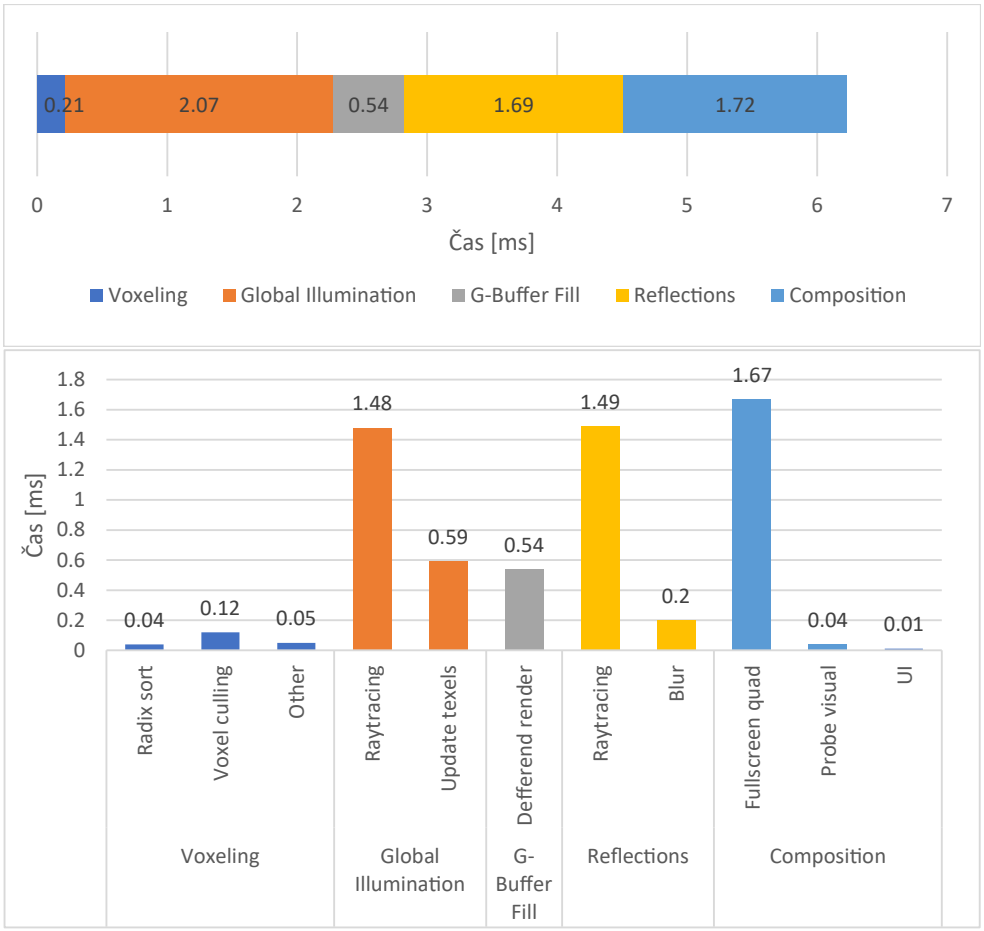
Tabuľka 6.1: Zostavy použité na meranie.

Jednotlivé merania boli uskutočnené pomocou nástroja *Nvidia Nsight*. Namerané časy by mali reprezentovať skutočnú prácu vykonanú na grafickej karte, resp. nezachycujú čas strávený na CPU. Typicky aplikácia spotrebuje okolo 1.4GB pamäte RAM a ďalších 1.4GB pamäte VRAM, z ktorej je približne 70% použitých len na textúry modelov (pri predvolenej scéne *Sponza*).

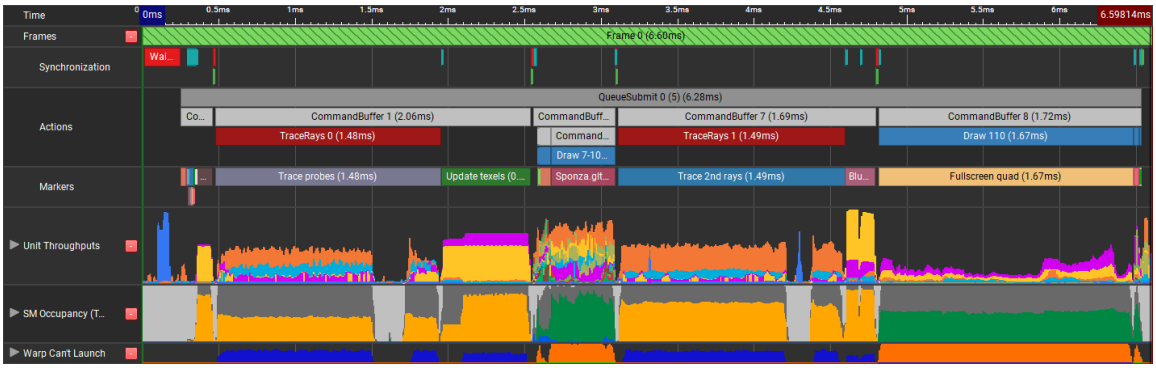
6.1.1 Meranie jednotlivých systémov

Meranie jednotlivých systémov (rendererov) implementovanej aplikácie bolo vykonané na scéne Sponza, vo *Full HD* rozlíšení a hustote $22 \times 10 \times 8$, resp. 1760 sond, ďalej z každej sondy bolo vysielaných 384 lúčov, z čoho 64 bolo statických. Toto predstavuje približne 6.7×10^5 vyslaných lúčov za jeden snímok, len pre globálnu iluináciu a ďalších 5.2×10^5 pri zapnutých odrazoch. Taktiež, každým snímkom sa raytracuje množstvo ďalších lúčov pri zisťovaní zatienenia daného fragmentu určitým svetlom. Namerané výsledky systémov popisuje graf 6.1.

Dôkladná analýza jedného snímku je zobrazená na obrázku 6.2. Z grafov a analýzy tohoto snímku je možno vidieť, že množstvo shaderov je limitovaných registrovou záťažou (spodná časť obrázku) – táto limitácia nemusí niekedy predstavovať až taký problém, pokiaľ daný shader nemá priveľa pamäťových latencií.



Obr. 6.1: Na grafoch sú zobrazené jednotlivé vykresľovacie systémy na scéne *Sponza*, vo *Full HD* rozlíšení a hustote sond $22 \times 10 \times 8$ pri 384 vysielaných lúčoch.



Obr. 6.2: Na obrázku je zobrazená analýza jedného snímku získaného pomocou programu *Nvidia Nsight – GPU trace*. V spodnej časti obrázku je znázornená SM okupancia – tmavo žltá, resp. zelená značí použité jednotky, zatiaľ čo tmavošedá značí SM jednotky, ktoré nemohli byť spustené a sivá značí nečinné SM jednotky. V spodnej časti je naznačené, prečo nemohli byť spustené (tmavošedé) SM jednotky – tmavo modrá zobrazuje nedostatok registrov pri *compute*, zatiaľ čo oranžová nedostatok registrov pri *fragment shaderoch*.

Avšak, pri oboch shaderoch, raytracovaní lúčov zo sond a finálnom vykresľovaní snímku, je nízka priepustnosť L1 pamätí cache, čo je v oboch prípadoch spôsobené nadmerným načítavaním textúr, resp. pri výpočte globálnej iluminácie je to spôsobené nekoherentnosťou lúčov (z každej sondy sú vysielané lúče iným smerom po celej scéne, čo spôsobuje, že geometria je zasahovaná kvázi stochasticky) a pri finálnom vykresľovaní snímku to je spôsobené nadmerným použitím textúr, resp. zlým návrhom tohoto shaderu.

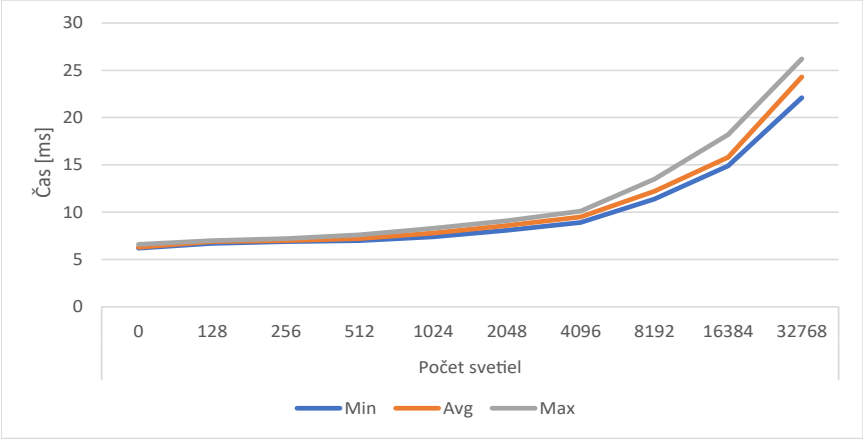
Možná oprava by spočívala v tom, že pri globálnej iluminácii by sa lúče mohli grupovať do skupín s rovnakým smerom a kvázi rovnakou pozíciou a finálne vykreslenie snímku by sa mohlo spraviť viac-priechodovo a tým by sa aj potenciálne zmenšila registrová záťaž, ktorá je čiastočne zvýšená použitím *ray queries* pre výpočet tieňov.

6.1.2 Vplyv počtu bodových svetiel na výkon

Testovanie bolo vykonané na scéne *Sponza* vo *Full HD* rozlíšení. Namerané hodnoty sú uvedené v tabuľke 6.2, resp. v grafe 6.3. Voxelizácia scény, resp. *light culling*, predstavoval pri počte 10^4 svetiel nárast o $1.71ms$, zatiaľ čo vykreslenie finálneho snímku zaznamenalo nárast o $2.75ms$. Voxelizácia bola dobre optimalizovaná a brzdená priepustnosťou L2 pamäte, zatiaľ čo pri vykreslení snímku sa prejavovali tie isté problémy popísané v sekcii 6.1.1.

Počet svetiel	Čas [ms]			
	Min	Avg	Max	Diff
0	6.2	6.3	6.6	0.0
128	6.7	6.9	7	+0.6
256	6.9	7	7.2	+0.7
512	7	7.2	7.6	+0.9
1024	7.4	7.8	8.3	+1.5
2048	8.1	8.6	9.1	+2.3
4096	8.9	9.5	10.1	+3.5
8192	11.4	12.2	13.5	+5.9
16384	14.9	15.8	18.2	+9.5
32768	22.1	24.3	26.2	+18.0

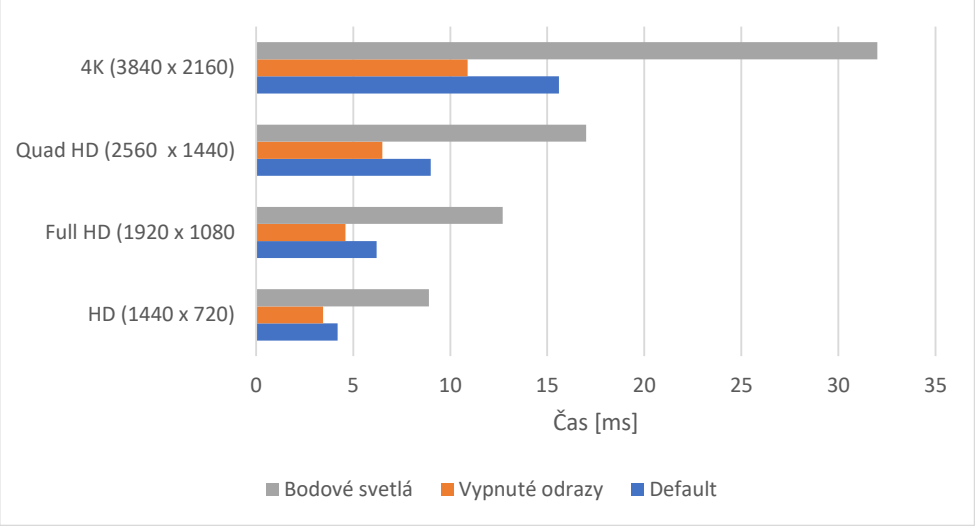
Tabuľka 6.2: Tabuľka vplyvu počtu svetiel na výkon.



Obr. 6.3: Graf zobrazujúci vplyv počtu svetiel na výkon.

6.1.3 Vplyv veľkosti rozlíšenia na výkon

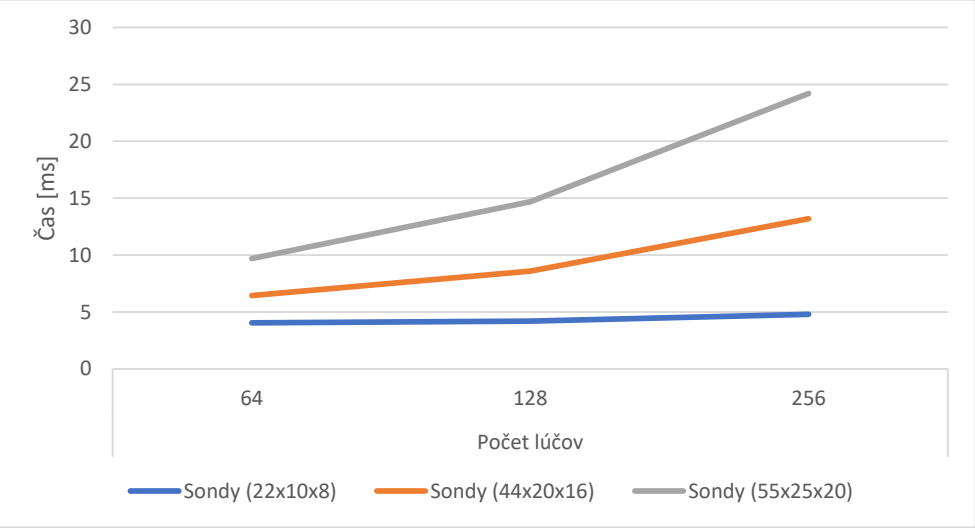
Testovanie vplyvu veľkosti rozlíšenia na rýchlosť výsledného renderu bolo uskutočnené na scéne *Sponza*. Porovnávané boli veľkosti rozlíšenia a taktiež aj vplyv určitých systémov, na ktoré vplýva zmena veľkosti rozlíšenia. Namerané hodnoty sú zobrazené v grafe 6.4.



Obr. 6.4: Vplyv zmeny rozlíšenia na výkonnosť aplikácie, taktiež vzhľadom na systémy, ktoré sú ovplyvňované zmenou rozlíšenia. V základe má scéna *Sponza* zapnuté odrazy a v scéne sa nachádza jedno bodové a jedno smerové svetlo. Pri testovaní na vplyv bodových svetiel sú odrazy opäť zapnuté a počet bodových svetiel je 10^4.

6.1.4 Vplyv počtu sond na výkon

Testovanie vplyvu počtu sond na výkon bolo testované na scéne *Sponza* pri *Full HD* rozlíšení.



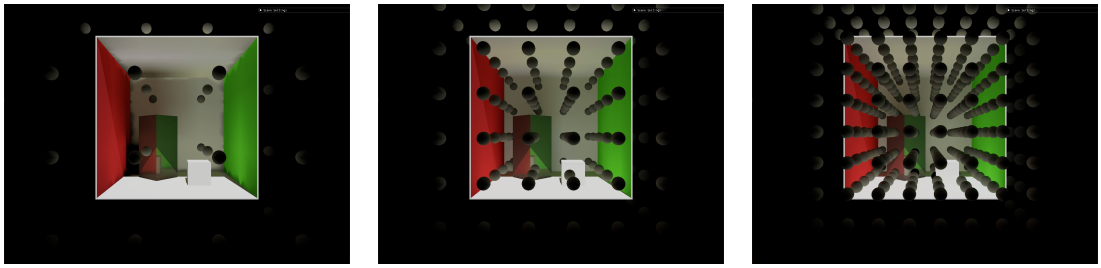
Obr. 6.5: Graf vplyvu počtu sond a počtu vysielaných lúčov na rýchlosť vykreslenia.

6.2 Porovnanie kvality osvetlenia

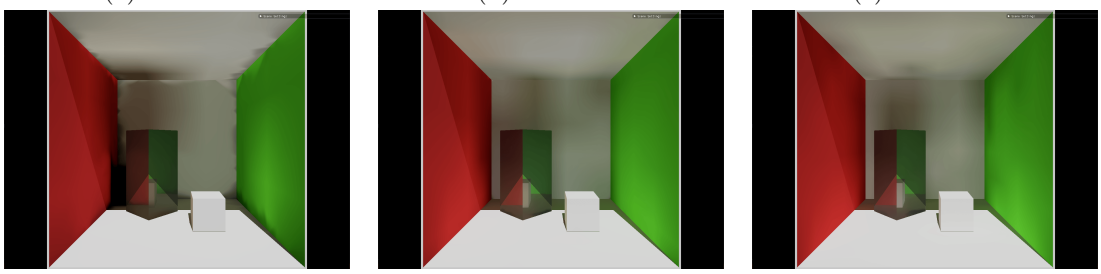
Porovnanie kvality osvetlenia bolo vykonané na testovacej scéne *Cornell box*. Testovaná bola zvyšujúca sa hustota sond voči zvyšujúcemu sa počtu vysielaných lúčov a výsledný vplyv na kvalitu daného osvetlenia (obr. 6.7). Z obrázkov je vidieť, že pri malej hustote sond vznikajú na okrajoch scény artefakty, resp. *light bleeding*. Pri zvyšujúcej sa hustote sond sa tento artefakt stráca. Počet vysielaných lúčov nemal až taký veľký vplyv na kvalitu osvetlenia, avšak pri nízkych počtoch lúčov scéna blikala, čo z obrázkov nie je možné vidieť.

Taktiež, pokiaľ je nejaká sonda zaseknutá v geometrii, vzniká inkorektné osvetlenie, resp. zatienenie tam, kde by avšak svetlo malo byť, čo je možné vidieť na obrázkoch 6.7d 6.7g a 6.7j v ľavom zadnom rohu.

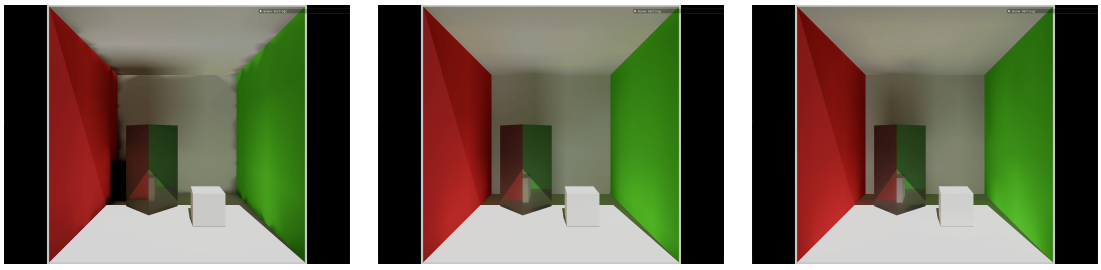
Obr. 6.6: Na obrázku je zobrazená scéna *Cornell box closed*, v ktorej vnútrajšku sa nenachádza žiaden zdroj svetla. Pri tenkých stenách, resp. ich spojoch vzniká *light bleeding*, aj pri použití *normal* a *view biasu*, pokiaľ je kamera v blízkosti jednej zo stien.



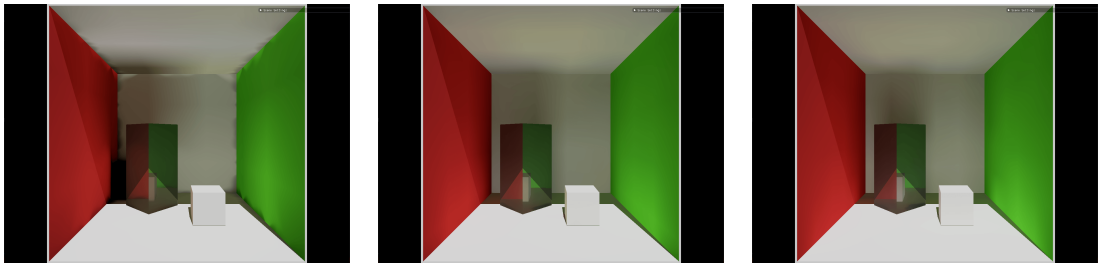
(a) $4 \times 4 \times 4$ (b) $6 \times 6 \times 6$ (c) $8 \times 8 \times 8$



(d) $4 \times 4 \times 4$, 64 lúčov (e) $6 \times 6 \times 6$, 64 lúčov (f) $8 \times 8 \times 8$, 64 lúčov



(g) $4 \times 4 \times 4$, 128 lúčov (h) $6 \times 6 \times 6$, 128 lúčov (i) $8 \times 8 \times 8$, 128 lúčov



(j) $4 \times 4 \times 4$, 256 lúčov (k) $6 \times 6 \times 6$, 256 lúčov (l) $8 \times 8 \times 8$, 256 lúčov

Obr. 6.7: V tomto obrázku je zobrazené porovnanie kvality osvetlenia so zvyšujúcou sa hustotou sond voči zvyšujúcemu sa počtu lúčov. Na obrázkoch **a** - **c** je zobrazené rozmiestnenie sond v testovacej scéne a potom obrázkoch **d** - **l** sú zobrazené jednotlivé osvetlenia scény s danou konfiguráciou.

6.3 Zhodnotenie

Metóda *Dynamic diffuse global illumination with raytraced irradiance fields* je relatívne lacná metóda na výpočet globálnej iluminácie. Táto metóda voči predchádzajúcej metóde umožňuje výpočet dynamickej globálnej iluminácie, teda globálna iluminácia nie je predpočítaná, ale počíta sa každý snímok, a teda dokáže zachytiť aj dynamickú geometriu a to v interaktívnych snímkoch za sekundu.

Táto metóda však trpí tzv. *light bleeding*, teda pretekaním svetla za geometriu, kde by mal byť tieň. Toto čiastočne opravujú *view* a *normal* biasy, avšak v niektorých prípadoch ani toto nestačí.

Pri implementácii viacero svetiel bola snaha optimalizovať počet vykresľovaných svetiel v jednom snímku pomocou stochastického prepínania medzi jednotlivými svetlami. Toto by mohlo fungovať pri výpočte globálnej iluminácie, pretože svetelný tok jednotlivých sond je zmiešaný medzi jednotlivými snímkami na základe histerézy, avšak pri výpočte priameho osvetlenia táto technika nefunguje (scéna bliká).

Typicky sa viacero svetiel implementuje taktiež stochasticky s tým, že sa neprepínajú jednotlivé svetlá, ale tieto svetlá sú vzorkované na základe *BRDF*, alebo inej funkcie. Vzniknutý snímok je ale veľmi zašumený a na odšumenie sa typicky používajú konvolučné neurónové siete.

Toto postačuje pre stotisícky svetiel, avšak pre väčší počet treba zvoliť iné techniky, ako napr. techniky použité v *RTXDI* [1] hlavne metóda *ReSTIR* [2]. Jej hlavná myšlienka je, že pokiaľ daný fragment osvetľuje veľa svetiel, jednotlivé tiene vrhané geometriou sú ľudským okom takmer nepovšimnuteľné a teda možno uplatniť dve veci: výpočet tieňov sa pre tento fragment môže úplne vynechať a z množiny svetiel osvetľujúcich daný fragment stačí vybrať len tie, ktoré prispievajú najviac.

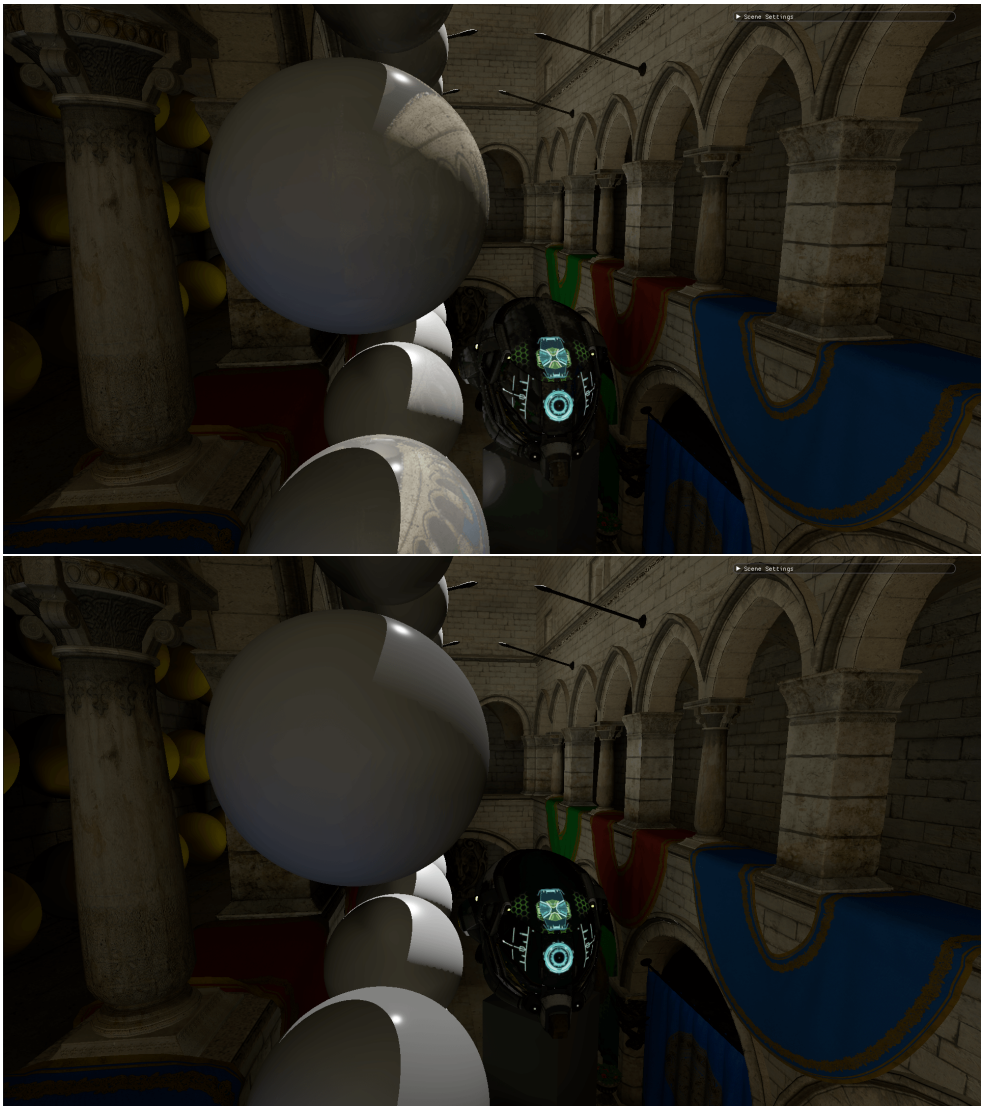
6.3.1 Možné vylepšenia metódy

Množstvo vylepšení bolo popísaných v nasledujúcej publikácii [26]. Azda najvýznamnejšie sú dynamické upravovanie heuristiky na základe zmeny svetelného toku v scéne, dynamické upravovanie pozície sond (pokiaľ je sonda zaseknutá v geometrii, je z nej posunutá von), stavový automat sond, ktorý umožňuje uspať jednotlivé sondy a tým zamedziť blikaniu scény, prípadne zlepšiť výkon. Za zmienku taktiež stojí hierarchia sond, kde v najnižšej úrovni sú jednotlivé sondy „prilepené“ na kameru, a táto klietka sond je veľmi hustá, čo zabezpečí kvalitnú globálnu iluináciu v okolí kamery.

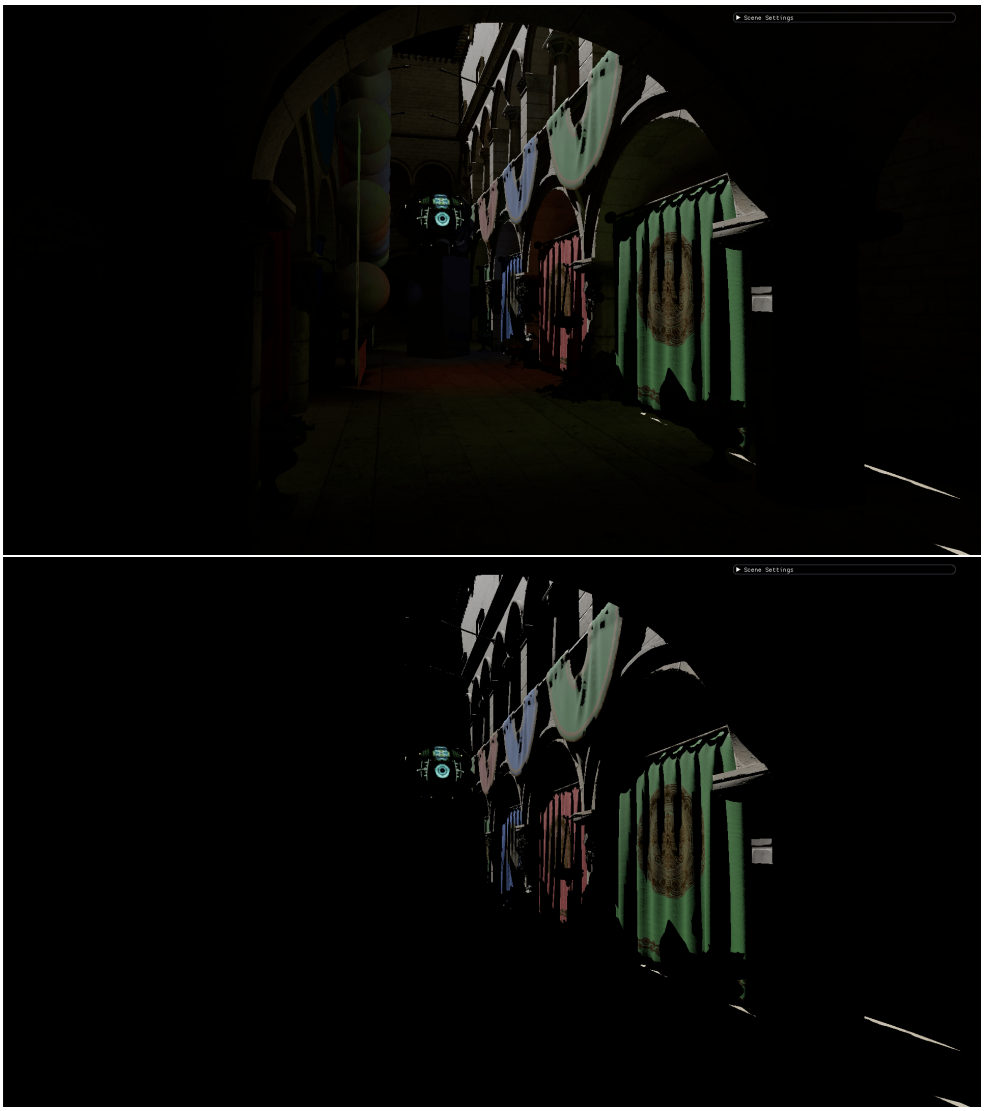
Tvorba BVH stromu svetiel a následný test na kolízie by mohli byť implementované pomocou hardvérového raytracingu. Avšak, bolo by nutné dôkladnejšie skúmanie, či by bol tento spôsob efektívnejší, keďže hardvérový raytracing je optimalizovaný najmä na kolízie s geometriou, resp. trojuholníkmi a implementácia BVH stromu svetiel (kapitola 4.4) v *compute* je optimalizovaná na analytickú geometriu a veľkosť BVH stromu.

[1] Dostupné na adrese https://developer.nvidia.com/rtxdi.

6.4 Grafický výstup aplikácie

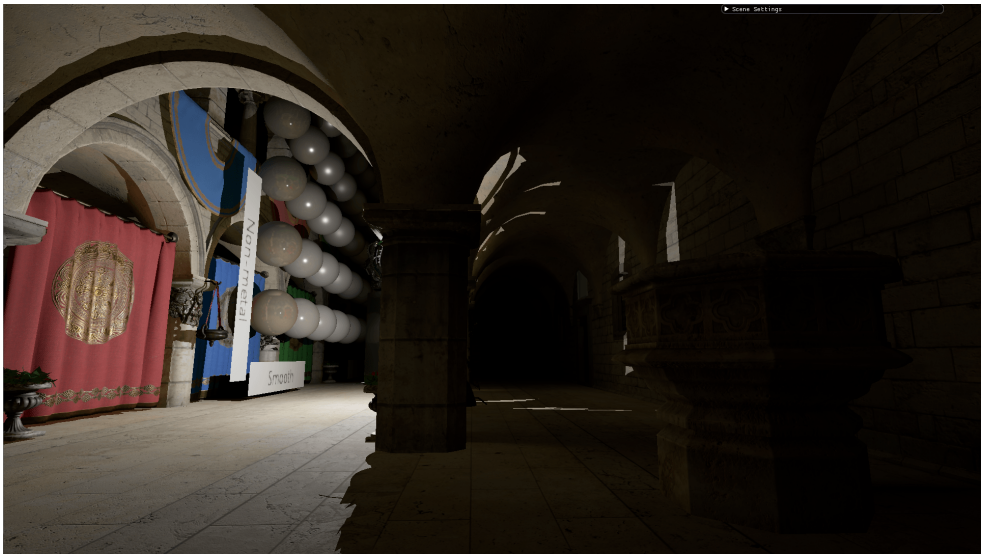


Obr. 6.8: Porovnanie zapnutých odrazov (obrázok hore) voči vypnutým (obrázok dole).



Obr. 6.9: Porovnanie scény obsahujúcej globálnu iluminániu (obrázok hore) voči scéne len s lokálnou iluminániou (obrázok dole).

Obr. 6.10: Zobrazenie scény *Sponza*, ktorá obsahuje 10^4 bodových zdrojov svetiel.



Obr. 6.11: Zobrazenie scény *Sponza*, ktorá je osvetlená jedným bodovým zdrojom svetla.

Kapitola 7

Záver

Cieľom tejto diplomovej práce bolo naštudovanie metód výpočtu globálnej iluminácie v reálnom čase a následná implementácia demonštračnej aplikácie na výpočet globálnej iluminácie v reálnom čase. Práca je rozdelená do niekoľkých logických celkov. V prvej časti sú popísané fyzikálne vlastnosti svetla a následne sú popísané používané metódy na výpočet globálnej iluminácie. V ďalšej časti bol popísaný návrh aplikácie a implementačné detaily so snahou na efektívnu utilizáciu hardvéru.

Výsledná aplikácia bola implementovaná úplne od základov v modernom grafickom aplikačnom rozhraní Vulkan a vybraná metóda bola *Dynamic diffuse global illumination with raytraced irradiance fields*. Táto metóda prináša dynamickú globálnu iluminániu v reálnom čase a je veľmi rýchla. Avšak trpí nedostatkom, tzv. *light bleeding* kedy svetlo preniká za geometriu, tam kde by nemalo. Ďalej táto metóda dokáže počítať globálnu iluminániu len z difúznych povrchov a teda spekulárne odrazy sa musia zanedbávať.

V aplikácii bolo implementovaných množstvo rozšírení ako napr. systém vykresľovania viacero bodových svetiel v reálnom čase a následná experimentácia a snaha o optimalizáciu tohoto systému. Ďalej *tone mapping* na prevod z HDR obrazu do LDR obrazu. Na vykresľovanie bola použitá pokročílá technika *PBR*, a taktiež bol implementovaný systém spekulárnych a glossy odrazov a posledne netriviálny a veľmi efektívny radix sort.

Literatúra

[1] Adinets, A. *A Faster Radix Sort Implementation*. 2020. Dostupné z: https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21572-a-faster-radix-sort-implementation.pdf.

[2] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A. et al. Spatiotemporal Reservoir Resampling for Real-Time Ray Tracing with Dynamic Direct Lighting. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. júl 2020, zv. 39, č. 4. ISSN 0730-0301. Dostupné z: https://doi.org/10.1145/3386569.3392481.

[3] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A. et al. Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. júl 2017, zv. 36, č. 4. DOI: 10.1145/3072959.3073601. ISSN 0730-0301. Dostupné z: https://doi.org/10.1145/3072959.3073601.

[4] Christian. *Followup: Normal Mapping Without Precomputed Tangents*. 2013. [Online; navštívené 23.4.2021]. Dostupné z: http://www.thetenthplanet.de/archives/1180.

[5] Cigolle, Z. H., Donow, S., Evangelakos, D., Mara, M., McGuire, M. et al. A Survey of Efficient Representations for Independent Unit Vectors. *Journal of Computer Graphics Techniques (JCGT)*. April 2014, zv. 3, č. 2, s. 1–30. ISSN 2331-7418. Dostupné z: http://jcgt.org/published/0003/02/01/.

[6] Cook, R. L. a Torrance, K. E. A Reflectance Model for Computer Graphics. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. január 1982, zv. 1, č. 1, s. 7–24. DOI: 10.1145/357290.357293. ISSN 0730-0301. Dostupné z: https://doi.org/10.1145/357290.357293.

[7] Corporation, N. *NVIDIA TURING GPU ARCHITECTURE Graphics Reinvented*, 14. September 2018. Dostupné z: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[8] Corporation, N. *NVIDIA AMPERE GA102 GPU ARCHITECTURE The Ultimate Play*, 16. September 2020. Dostupné z: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

[9] Crassin, C., Neyret, F., Sainz, M., Green, S. a Eisemann, E. Interactive indirect illumination using voxel cone tracing. In: Wiley Online Library. *Computer Graphics Forum*. 2011, sv. 30, č. 7, s. 1921–1930. Dostupné z: https://doi.org/10.1111/j.1467-8659.2011.02063.x.

[10] Dachsbacher, C. a Stamminger, M. Reflective Shadow Maps. In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: Association for Computing Machinery, 2005, s. 203–231. I3D '05. ISBN 1595930132. Dostupné z: https://doi.org/10.1145/1053427.1053460.

[11] Donnelly, W. a Lauritzen, A. Variance Shadow Maps. In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: Association for Computing Machinery, 2006, s. 161–165. I3D '06. DOI: 10.1145/1111411.1111440. ISBN 159593295X. Dostupné z: https://doi.org/10.1145/1111411.1111440.

[12] Goral, C. M., Torrance, K. E., Greenberg, D. P. a Battaile, B. Modeling the Interaction of Light between Diffuse Surfaces. *SIGGRAPH Comput. Graph.* Association for Computing Machinery. január 1984, zv. 18, č. 3, s. 213–222. ISSN 0097-8930. Dostupné z: https://doi.org/10.1145/964965.808601.

[13] Haines, E. a Akenine Möller, T., ed. *Ray Tracing Gems*. Apress, 2019. http://raytracinggems.com.

[14] Jensen, H. W. Global Illumination Using Photon Maps. In: *Proceedings of the Eurographics Workshop on Rendering Techniques '96*. Berlin, Heidelberg: Springer-Verlag, 1996, s. 21–30. ISBN 3211828834.

[15] Jensen, H. W. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2001. ISBN 1568811470.

[16] Kajiya, J. T. The Rendering Equation. New York, NY, USA: Association for Computing Machinery. august 1986, zv. 20, č. 4, s. 143–150. DOI: 10.1145/15886.15902. ISSN 0097-8930. Dostupné z: https://doi.org/10.1145/15886.15902.

[17] Kaplanyan, A. Light Propagation Volumes in CryEngine 3. In:. Január 2009.

[18] Karas, M. *Clustered deferred sahding vo Vulkan API*. Brno, 2019. Bakalárska práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedúci práce Milet, T. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/122093?zp_id=122093.

[19] Karis, B. *Real Shading in Unreal Engine 4*. Epic Games, 2013. Dostupné z: http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf.

[20] Karras, T. Thinking Parallel, Part III: Tree Construction on the GPU. Nvidia.com. December 2012. Dostupné z: https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-construction-gpu/.

[21] Keller, A. Instant Radiosity. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co., 1997, s. 49–56. SIGGRAPH '97. ISBN 0897918967. Dostupné z: https://doi.org/10.1145/258734.258769.

[22] Khronos. *Vulkan specification (with all refistered Vulkan extensions)*, 22. November 2020. Dostupné z: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/index.html.

[23] Kirk, D. *Graphics Gems III*. AP Professional, 1992. Graphics gems series : a collection of practical techniques for the computer graphics programmer. ISBN 9780124096738.

[24] Laine, S., Saransaari, H., Kontkanen, J., Lehtinen, J. a Aila, T. Incremental Instant Radiosity for Real-Time Indirect Illumination. In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. Eurographics Association, 2007, s. 277–286. EGSR'07. ISBN 9783905673524.

[25] Majercik, Z., Guertin, J.-P., Nowrouzezahrai, D. a McGuire, M. Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields. *Journal of Computer Graphics Techniques (JCGT)*. June 2019, zv. 8, č. 2, s. 1–30. ISSN 2331-7418. Dostupné z: http://jcgt.org/published/0008/02/01/.

[26] Majercik, Z., Marrs, A., Spjut, J. a McGuire, M. *Scaling Probe-Based Real-Time Dynamic Global Illumination for Production*. 2020.

[27] Mallett, I. a Yuksel, C. Constant-time energy-normalization for the Phong specular BRDFs. *The Visual Computer*. Oct 2020, zv. 36, č. 10, s. 2029–2038. DOI: 10.1007/s00371-020-01954-x. ISSN 1432-2315. Dostupné z: https://doi.org/10.1007/s00371-020-01954-x.

[28] McGuire, M. *DDGI: Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields (I3D Presentation Slides)* [Journal of Computer Graphics Techniques]. 2019. Dostupné z: http://jcgt.org/published/0008/02/01/.

[29] McGuire, M., Mara, M., Nowrouzezahrai, D. a Luebke, D. Real-Time Global Illumination Using Precomputed Light Field Probes. In: *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: Association for Computing Machinery, 2017. I3D '17. DOI: 10.1145/3023368.3023378. ISBN 9781450348867. Dostupné z: https://doi.org/10.1145/3023368.3023378.

[30] Meister, D., Boksansky, J., Guthe, M. a Bittner, J. On Ray Reordering Techniques for Faster GPU Ray Tracing. In: *Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: Association for Computing Machinery, 2020. I3D '20. DOI: 10.1145/3384382.3384534. ISBN 9781450375894. Dostupné z: https://doi.org/10.1145/3384382.3384534.

[31] Merrill, D. a Garland, M. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002*. 2016.

[32] Nguyen, H. *GPU Gems 3*. Addison-Wesley Professional, 2007. ISBN 9780321515261.

[33] Olsson, O. a Assarsson, U. Tiled shading. *Journal of Graphics, GPU, and Game Tools*. Taylor & Francis. 2011, zv. 15, č. 4, s. 235–251.

[34] Olsson, O., Billeter, M. a Assarsson, U. Clustered Deferred and Forward Shading. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. Goslar Germany, Germany: Eurographics Association, 2012, s. 87–96. EGGH-HPG'12. DOI: 10.2312/EGGH/HPG12/087-096. ISBN 978-3-905674-41-5. Dostupné z: https://doi.org/10.2312/EGGH/HPG12/087-096.

[35] Pharr, M., Jakob, W. a Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Elsevier Science, 2016. ISBN 9780128007099. Dostupné z: https://books.google.sk/books?id=iNMVBQAAQBAJ.

[36] Schlick, C. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum*. 1994, zv. 13, s. 233–246.

[37] Tatzgern, W., Mayr, B., Kerbl, B. a Steinberger, M. Stochastic Substitute Trees for Real-Time Global Illumination. In: *Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: Association for Computing Machinery, 2020. I3D '20. DOI: 10.1145/3384382.3384521. ISBN 9781450375894. Dostupné z: https://doi.org/10.1145/3384382.3384521.

[38] Veach, E. a Guibas, L. J. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In: *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1995, s. 419–428. SIGGRAPH '95. DOI: 10.1145/218380.218498. ISBN 0897917014. Dostupné z: https://doi.org/10.1145/218380.218498.

[39] Vries, J. de. PBR Theory. *Learnopengl* [online]. Dostupné z: https://learnopengl.com/PBR/Theory. Path: PBR; Theory.

[40] Vries, J. D. *Normal Mapping*. 2014. [Online; navštívené 23.4.2021]. Dostupné z: https://learnopengl.com/Advanced-Lighting/Normal-Mapping.

[41] Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M. et al. Lightcuts: A Scalable Approach to Illumination. New York, NY, USA: Association for Computing Machinery. júl 2005, zv. 24, č. 3, s. 1098–1107. DOI: 10.1145/1073204.1073318. ISSN 0730-0301. Dostupné z: https://doi.org/10.1145/1073204.1073318.

[42] Woop, S., Benthin, C. a Wald, I. Watertight Ray/Triangle Intersection. *Journal of Computer Graphics Techniques (JCGT)*. June 2013, zv. 2, č. 1, s. 65–82. ISSN 2331-7418. Dostupné z: http://jcgt.org/published/0002/01/05/.

[43] Zhang, Z. a Shafer, D. Introductory Statistics Shafer and Zhang. 2016.